END-TO-END CONCURRENT MULTIPATH TRANSFER USING TRANSPORT LAYER MULTIHOMING

by

Janardhan R. Iyengar

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Summer 2006

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
End-to-End Comcurrent Multipath Transfer Using Transport Layer Multihoming	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Delaware,Computer and Information Sciences Department,Newark,DE,19716**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**123**	

END-TO-END CONCURRENT MULTIPATH TRANSFER USING TRANSPORT LAYER MULTIHOMING

by

Janardhan R. Iyengar

Approved: _____

B. David Saunders, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Thomas M. Apple, Ph. D.
Dean of the College of Arts and Sciences

Approved: _____

Daniel Rich, Ph.D.
Provost

To Anna (Sudarshan)

ACKNOWLEDGMENTS

I am fortunate to have had Professor Paul Amer as my Ph.D. advisor. His ability to ask the right questions and his attention to detail have never ceased to surprise me. He was always able to make time for me, even if I just walked into his office without any notice. His constant emphasis on clear communication and presentation have helped me significantly. Over the years, as I have gained a deeper appreciation of an advisor's responsibilities, my respect for Prof. Amer has only grown. I have a lot to thank him for.

I thank my dissertation committee members: Prof. Adarshpal Sethi, Prof. Stephan Bohacek, Prof. Phillip Conrad, and Randall Stewart for providing valuable direction and feedback. I specifically want to thank Randy for travelling great distances to be on my committee, and spending much time with me over the years. His inputs and insights have been invaluable.

I have had the good fortune of interacting and working with some really wonderful people in the Protocol Engineering Lab (PEL) and in the CIS department, and I thank them all. Armando Caro, Jerry Heinz, Sourabh Ladha, Keyur Shah, Len Armstrong, Mark Hufe, Preethi Natarajan and Jon Leighton have all made my years at PEL a very enjoyable time.

Two people stand out in my tenure at UD, for whose friendships I am thankful. Armando Caro, a wonderful person and researcher, and my friend of several years, has always been great to be around and work with. I've enjoyed our innumerable long discussions and intense brainstorming / coding / debugging / racing-for-the-paper-deadline sessions. I could always count on him as a friend (and chauffeur!), especially so in times of need.

Phill Conrad, my friend, philosopher and guide, has enriched my life in many significant ways. From helping me immensely with my job search and interviews to discussing my life's sketches and dilemmas, he was always there to help out despite his busy schedule. I thank Phill from the bottom of my heart for being him. Phill and Bob have been wonderful friends to my family, and I thank them for the wonderful times they have spent with us.

I am very grateful to have a very loving and understanding family. Amma and Appa have somehow managed the incredible patience to hang on to the hope that I will, some day, take my "final exams" and graduate. I hope to make up the time that I have been unable to spend with them and with my lovely sister, Vanu. I also thank my Anna and Manni for always being with me, especially in difficult times.

My closest friend and wife, Laura, and my son, Achintya, have both been blessings. I am deeply thankful to them for putting up with me, including my consistently predictable "I'm sorry I haven't left yet" each evening. I could not ask for more than to have them (and the one about to join us) as the center of my life.

A special note for my darling older brother (Anna), Sudarshan, who, in incredibly difficult times, made certain that I continued on with my Ph.D. I cannot express my gratitude or my love for him in mere words. This dissertation is dedicated to him.

All the dear friends who have been with me during this period have made all these years wonderful and memorable.

Finally, my doctoral work was funded by: (i) the U.S. Army Research Laboratory under the Federated Laboratory Program and the Collaborative Technology Alliance Program, (ii) the University of Delaware through two Competitive Fellowships and one Dissertation Fellowship, and (iii) the University Research Program of Cisco Systems, Inc. My thanks to all for their generous funding that made this research possible.

TABLE OF CONTENTS

ABSTRACT

Transport layer multihoming binds a single transport layer association to multiple network addresses at each endpoint, thus allowing the two end hosts to communicate over multiple network paths. This dissertation investigates end-to-end *Concurrent Multipath Transfer (CMT)* using transport layer multihoming for increased application throughput. CMT is the simultaneous transfer of new data from a source host to a destination host via two or more end-to-end paths. We investigate and evaluate design considerations in implementing CMT at the transport layer using the Stream Control Transmission Protocol (SCTP) as an example of a multihome-capable transport layer protocol. Specifically, we explore (i) algorithms for CMT at the transport layer, (ii) retransmission policies for CMT, and (iii) performance implications of a bounded receive buffer on CMT.

We identify three negative side-effects of reordering due to CMT that must be managed before the full performance gains of CMT's parallel transfer can be achieved. We propose three algorithms to eliminate these side-effects: the Split Fast Retransmit algorithm (SFR) to handle unnecessary fast retransmissions by a sender, the Cwnd Update for CMT algorithm (CUC) to counter overly conservative congestion window growth at a sender, and the Delayed Ack for CMT algorithm (DAC) to curb an increase in ack traffic due to fewer delayed acks by a receiver. These algorithms demonstrate that a single sequence space within a transport layer association is sufficient for CMT; separate sequence spaces per path are not required.

We propose and evaluate five retransmission policies for CMT. Introducing these retransmission policies causes two side-effects: occurrence of spurious retransmissions and inaccurate congestion window estimation. We propose two protocol modifications to eliminate these side-effects. Using simulation, we evaluate CMT against *AppStripe*, a simulated idealized application that stripes data over multiple paths using multiple SCTP associations. The results of this evaluation demonstrate that CMT's sharing of sequence space across paths improves performance—an inherent benefit that load sharing at the transport layer has over that at the application layer.

We study the performance of CMT in the presence of a bounded receive buffer (rbuf). Simulation results show that if two paths are used for CMT, the lower quality (i.e., higher loss rate) path degrades overall throughput of an rbuf-constrained CMT association by blocking the rbuf. We argue that rbuf blocking is not specific to the transport layer, but applies to multipath transfers at other layers as well. We present and discuss CMT performance using the proposed retransmission policies and various constrained rbuf values. We also study the impact of rbuf blocking with different combinations of end-to-end loss rate and delay on the two paths and show that when large differences exist in path delays and loss rates, using only the better path outperforms using two paths concurrently. While rbuf blocking cannot be eliminated, we show that it can be reduced by choice of retransmission policy—a mechanism available to only the transport layer. We recommend the loss-rate-based policies, which are the best performing ones, for CMT.

We discuss our implementation of CMT in the reference SCTP implementation, which is written for the BSD family of operating systems. This implementation effort was funded by Cisco Systems, with the goal of potentially migrating CMT into their systems. While we test our implementation in the FreeBSD operating system, other BSD and BSD-derived systems such as NetBSD, OpenBSD and Darwin should be able to use the CMT implementation. A major goal of our implementation is to encourage wider use

and experimentation with CMT in different environments and under different constraints, contributing to a better understanding of CMT and to uncovering of hitherto unknown issues.

This dissertation operates under the strong assumption that the bottleneck queues used in CMT are independent. We identify two classes of networks where this assumption is valid, and thus the results in this dissertation can be of immediate application. We also explain how this dissertation lays the foundation for further exploration of CMT for the Internet, where this assumption does not hold.

Chapter 1

INTRODUCTION

1.1 Problem Statement

This dissertation proposes and investigates transport layer techniques to exploit end host multihoming using end-to-end Concurrent Multipath Transfer (CMT) for improved application throughput. CMT is the concurrent transfer of new data from a source to a destination host via two or more end-to-end paths. A host is multihomed if it can be addressed by multiple IP addresses [15], as is the case when the host has multiple network interfaces. Transport layer support of multihoming can allow, transparent to the application (or users), dynamic redirection of data within a transport layer connection to a reachable destination address during network congestion and/or failure. With the inclusion of multihoming support in new and practical transport protocols being standardized by the IETF, the need to research new techniques to meaningfully and correctly leverage multihoming has become immediate. Towards this end, we propose to extend the benefits of transport layer multihoming by distributing data over the multiple end-to-end paths between a multihomed source and destination, and discuss the different issues therein.

1.2 Motivation

Multihoming among networked machines and devices is a technologically feasible and increasingly economical proposition. Though feasibility alone does not determine adoption of an idea, multihoming can be expected to be the rule rather than the exception in the near future, particularly for applications where fault tolerance is crucial. Multihomed nodes may be simultaneously connected through multiple access technologies, and even multiple end-to-end paths to increase resilience to path failure. For instance, users may be simultaneously connected through dial-up/broadband, or via multiple wireless technologies such as 802.11b and GPRS. (For a more detailed motivation of end host multihoming, see Chapter 1 in [16].)

Although transport layer multihoming is an old concept (splitting/recombining or downward-multiplexing for providing added resilience against network failure and/or potentially increasing throughput [32, 66, 69]), the current transport protocol workhorses, UDP and TCP, do not support multihoming; UDP does not bind or connect endpoints, and TCP allows binding to only one network address at each end of a connection. At the time TCP was designed, network interfaces were expensive components, and hence multihoming of end hosts was beyond the ken of research. Changing economics and an increased emphasis on end-to-end fault tolerance have brought multihoming within the purview of the transport layer, thus transforming transport layer multihoming into a feature worth supporting and investigating.

Two recent IETF transport layer protocols, the Stream Control Transmission Protocol (SCTP) [63, 65], and the Datagram Congestion Control Protocol (DCCP) [49] natively support multihoming at the transport layer. While DCCP provides "primitive multihoming" [48] at the transport layer for mobility support, SCTP multihoming is driven by a broader and more generic application base, which includes fault tolerance and mobility. DCCP multihoming, by design, is useful only for connection migration and not CMT.

Simultaneous transfer of new data to multiple destination addresses is inherent in SCTP's design, but currently not allowed due primarily to insufficient research. This dissertation attempts to provide that needed research.

In this dissertation, we focus on SCTP for our investigation of CMT primarily due to lack of mature multihoming mechanisms in any other practical transport layer protocol, and partly due to this author's expertise with it. While concurrency can be arranged at other layers (as discussed in Chapters 3 and 7), we propose CMT at the transport layer because being the first end-to-end layer, the transport layer has the best knowledge to estimate end-to-end paths' characteristics. Though CMT uses SCTP in our analysis, our goal is to study CMT at the transport layer in general. We believe that the issues and algorithms considered in this research will provide insight in incorporating multihome-awareness in other transport protocols.

1.3 An SCTP Primer

We overview several ideas and mechanisms used by SCTP relevant to this dissertation; some are compared with TCP to highlight similarities and differences. SCTP was originally developed to carry telephony signaling messages over IP networks. With continued work, SCTP evolved into a general purpose transport protocol that includes advanced delivery options. SCTP is defined in RFC2960 [65] with changes and additions included in the Specification Errata Internet draft[1] [63]. Similar to TCP, SCTP provides a reliable, full-duplex connection, called an *association*. An SCTP packet, or more generally, protocol data unit (PDU), consists of one or more concatenated building blocks called *chunks*: either control or data. For the purposes of reliability and congestion control, each data chunk in an association is assigned a unique Transmission Sequence Number

[1] RFC2960 and the Specification Errata are expected to soon be combined into a single "RFC2960-bis" document.

(TSN). Since chunks are atomic, TSNs are associated with chunks of data, as opposed to TCP which associates a sequence number with each data octet in the bytestream. In our simulations, we assume one data chunk per PDU for ease of illustration; each PDU thus carries, and is associated with a single TSN. SCTP uses a selective ack scheme similar to SACK TCP [26]; SCTP acks carry cumulative and selective ack (also called *gap ack*) information and are called SACKs. In this dissertation, sometimes "SACK" is used rather than "ack" to emphasize when an ack carries both cumulative and selective ack information.

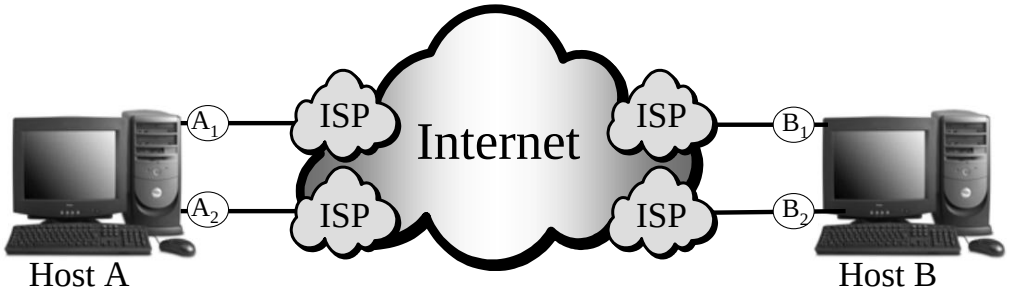


Figure 1.1: Example Multihomed Topology

Unlike TCP, SCTP supports multihoming at the transport layer to provide redundancy at the path level, thus increasing association survivability in the case of a network path failure. An SCTP endpoint may bind to multiple IP addresses during association initialization. Referring to Figure 1.1, let us contrast SCTP with TCP to further explain SCTP's multihoming feature. Four distinct TCP connections are possible between Hosts A and B: (A_1, B_1), (A_1, B_2), (A_2, B_1), (A_2, B_2). SCTP, on the other hand, is not forced to choose a single IP address on each host. Instead, a single SCTP association could consist of two sets of IP addresses, which in our example would be: $(\{A_1, A_2\}, \{B_1, B_2\})$. An SCTP endpoint may bind to a proper subset of its available IP addresses. This binding allows an SCTP sender to send data to a multihomed receiver through different destination addresses. Currently, SCTP uses multihoming for redundancy purposes only; the RFC2960

specification does not allow simultaneous transfer of *new* data to multiple destination addresses. New data must be sent to a single *primary destination*, while retransmissions may be sent to any alternate destination. Note that a single port number is used at each endpoint regardless of the number of IP addresses.

SCTP's congestion control algorithms are based on RFC 2581 [8], and include SACK-based mechanisms for better performance. Similar to TCP, SCTP uses three control variables: a receiver's advertised window (rwnd), a sender's congestion window (cwnd), and a sender's slow start threshold (ssthresh). However, unlike TCP's cwnd which reflects which and how much data can be sent, SCTP's cwnd dictates only how much data can be sent. Since an SCTP association allows multihomed source and destination endpoints, a source maintains several parameters on a *per destination* basis: cwnd, ssthresh, and roundtrip time (RTT) estimates. An SCTP sender also maintains a separate retransmission timer per destination, but rwnd is shared across an association.

A note on language and terminology. A reference to "cwnd for destination X" means the cwnd maintained at the sender for destination X, and "timeout on destination X" refers to the expiration of a sender's retransmission timer maintained for data sent to destination X. Also, "cwnd for destination X" may be used interchangeably with "cwnd for path Y", where path Y ends in destination X.

For the interested reader, more information on SCTP is available in [20, 43, 62–65].

1.4 Assuming Independent Bottlenecks

This dissertation operates under the assumption that the bottleneck queues (in other words, points of congestion) on the end-to-end paths used in CMT are independent. Overlap in the paths is acceptable as long as the paths' bottlenecks are independent. Two motivating examples where this assumption holds are telephony networks and battlefield networks.

- Signaling communication in telephony networks is being migrated to IP, and uses SCTP for transport. Given the stringent availability requirements on these networks, signaling devices are multihomed and are inter-connected via multiple, independent end-to-end paths for reasons of fault tolerance. The end-to-end paths share no network resource, thus avoiding any single point of failure [28].

- The US Army's proposed Future Combat System for battlefield networks will equip mobile hosts with multiple interfaces, often connecting to independent wireless networks, for example, a terrestrial short-range radio, and a long-range communication to either low-flying or geostationary satellites. These different communication technologies will provide multiple independent paths between nodes [1].

We recognize that our assumption of independent paths is a strong one. A CMT sender that can deal with shared bottlenecks must be able to detect them and respond appropriately. In Section 7.2.2, we discuss how related work on end-to-end shared bottleneck detection can be applied to CMT. In Section 7.1.3, we examine how the results of this dissertation are applicable to the further study of CMT in the presence of a shared bottleneck.

1.5 Research Overview

This dissertation proposes mechanisms for CMT using SCTP multihoming and investigates performance gains to be had with CMT. A structural outline of this dissertation is shown in Figure 1.2. The references cited for each chapter represent the author's publications for each topic.

A naïve form of CMT can be obtained by simply modifying the SCTP sender to transmit new data to all destinations. However, transmitting new data over multiple paths with different network characteristics introduces reordering in the data stream. *We identify*

three negative side-effects of reordering that need to be addressed, and propose three algorithms as solutions for a "correct" CMT—the Split Fast Retransmit algorithm (SFR) to handle unnecessary fast retransmissions by a sender, the Cwnd Update for CMT algorithm (CUC) to counter overly conservative congestion window growth at a sender, and the Delayed Ack for CMT algorithm (DAC) to curb an increase in ack traffic due to fewer delayed acks by a receiver. Details of the algorithms are presented in Chapter 2.

Though an SCTP sender can choose from one of several destinations to retransmit lost transmissions, a sender has limited feedback about the different paths to the receiver since new data is sent to only one destination. A CMT sender, on the other hand, has more frequent feedback about all paths to the receiver, and can leverage this information in making a better decision when retransmitting. A CMT sender can also choose the retransmission destination for each loss recovery independently using the most recent information for its decision. We explore several retransmission policies for CMT, and evaluate their performance against an idealized application that stripes data across multiple paths. Unexpectedly, we observe that *a key benefit with CMT is increased resilience to reverse path loss due to the use of multiple reverse paths*. These results are presented and discussed in Chapter 3.

We study the implications of a bounded receive buffer (rbuf)—a constraint often ignored in transport layer research—on CMT's performance. We observe that a bounded rbuf has significant impact on CMT's performance; if two paths are used for CMT, the lower quality (i.e., higher loss rate) path degrades the overall throughput of an rbuf-constrained CMT association. We study the impact and extent of this degradation, and observe that when large differences exist in path delays and loss rates, using only the better path outperforms using two paths concurrently. We argue that *this degradation is not specific to the transport layer, but applies to multipath transfers at other layers as well. While*

this degradation cannot be eliminated, it can be reduced by intelligent choice of retransmission policy—a mechanism available to only the transport layer. These results are presented in Chapter 4.

We discuss our implementation of CMT in the reference SCTP implementation, which is written for the BSD family of operating systems, and evaluate it over an emulated network. Our experimental results demonstrate the correctness of the implementation, and help validate our simulation results. Chapter 5 carries a discussion of the implementation and the experiments. While our implementation is tested in the FreeBSD operating system, other BSD and BSD-derived systems such as NetBSD, OpenBSD and Darwin should be able to use the CMT implementation. This implementation effort was funded by Cisco Systems, with the goal of potentially migrating CMT into their IOS operating system. This implementation should also encourage wider use and experimentation with CMT in varied environments; for instance, this author has been in touch with a group at the University of British Columbia who are interested in using the CMT implementation in grid networks. Such varied use will contribute to a better understanding of CMT and to uncovering of hitherto unknown issues.

In Chapter 6, we change topics to explore *changeover* with SCTP, where a sender redirects traffic from one reachable destination to another, only once in an association. We uncover a problem in SCTP that results in unnecessary retransmissions and excessively aggressive growth of the sender's congestion window (cwnd) during certain changeover conditions. Since CMT can be viewed as *repeated changeover*, we see similarity between this problem and one negative side-effect of reordering with CMT; indeed, the same algorithm (SFR) is proposed as a solution to both problems. To gain insight into the ambient conditions under which cwnd overgrowth can be observed with SCTP, we develop an analytical model of this behavior and analyze the results.

Chapter 7 concludes this dissertation. We first present a discussion of some design considerations for CMT and applicability of CMT in various settings. We then identify and present thoughts on areas for continued exploration of CMT, some of which are currently being pursued. Finally, we conclude with a summary of related work on load sharing in computer networking.

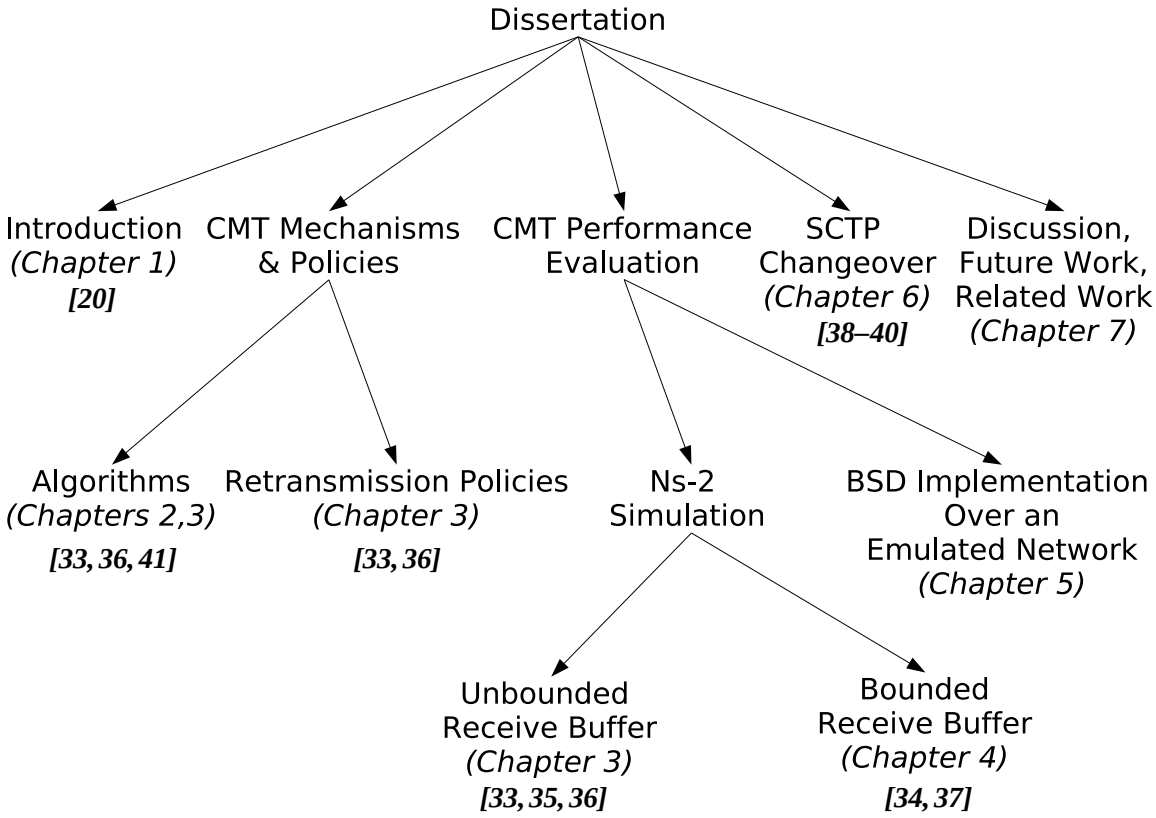


Figure 1.2: Dissertation structure

Chapter 2

CMT ALGORITHMS

As is the case with TCP [13, 14, 71], reordering introduced in an SCTP flow degrades throughput. When multiple paths being used for CMT have different delay and/or bandwidth characteristics, significant packet reordering can be introduced in the flow by a CMT sender. Reordering is a natural consequence of CMT, and is difficult to eliminate in an environment where the end-to-end path characteristics are changing or unknown *a priori*, as in the Internet. In this chapter, we identify and resolve the negative side-effects of sender-introduced reordering by CMT in SCTP. The results have been published in [33, 36, 41].

2.1 Naïve CMT: Reordering concerns

To illustrate the effects of reordering introduced in SCTP by CMT, we use the simple simulation setup in Figure 2.1. Two dualhomed hosts, sender A with local addresses A_1, A_2, and receiver B with local addresses B_1, B_2, are connected by two independent paths: Path 1 ($A_1 - B_1$), and Path 2 ($A_2 - B_2$) having end-to-end available bandwidths 0.2 Mbps and 1 Mbps, respectively. The roundtrip propagation delay on both paths is 90 ms, roughly reflecting the U. S. coast-to-coast delay. CMT sender A sends new data to destinations B_1 and B_2 concurrently, as bandwidth becomes available on corresponding paths, i.e., as corresponding cwnds allow. When cwnd space is available simultaneously

for two or more destinations, data is sent to these destinations in arbitrary order—a reasonable transmission policy when the CMT sender has no *a priori* knowledge of the paths' characteristics.

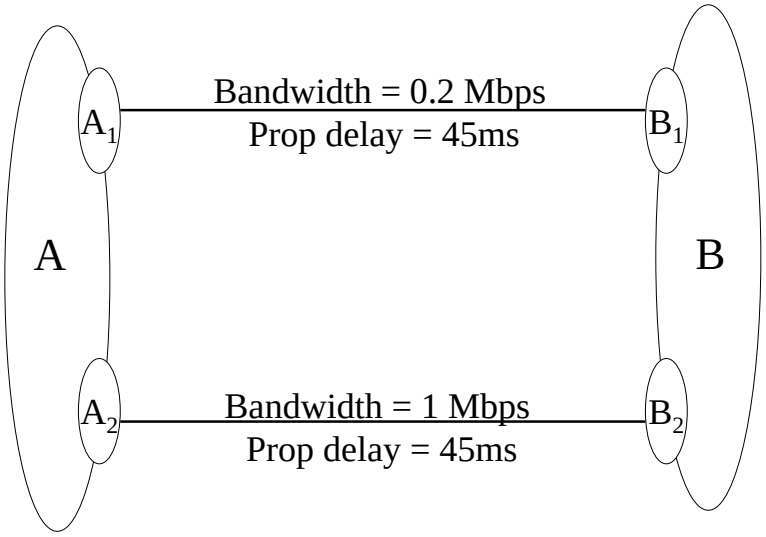


Figure 2.1: Simulation topology to illustrate reordering effects

The simulation results described in this chapter (Figures 2.2 and 2.6) both show cwnd evolution over time. The figures show the CMT sender's (1) observed cwnd evolution for destination B_1 ($+$), (2) observed cwnd evolution for destination B_2 ($\times$), (3) calculated aggregate cwnd evolution (sum of (1) and (2)) ($\triangle$), and (4) expected aggregate cwnd evolution (–). This last curve represents our initial performance goal for CMT—the sum of the cwnd evolution curves of two independent SCTP runs, using B_1 and B_2 as the primary destination, respectively.

Figure 2.2 shows how, when performing CMT using SCTP without any modifications (i.e., Naïve CMT), reordering significantly hinders both B_1 and B_2's cwnd growth. Note

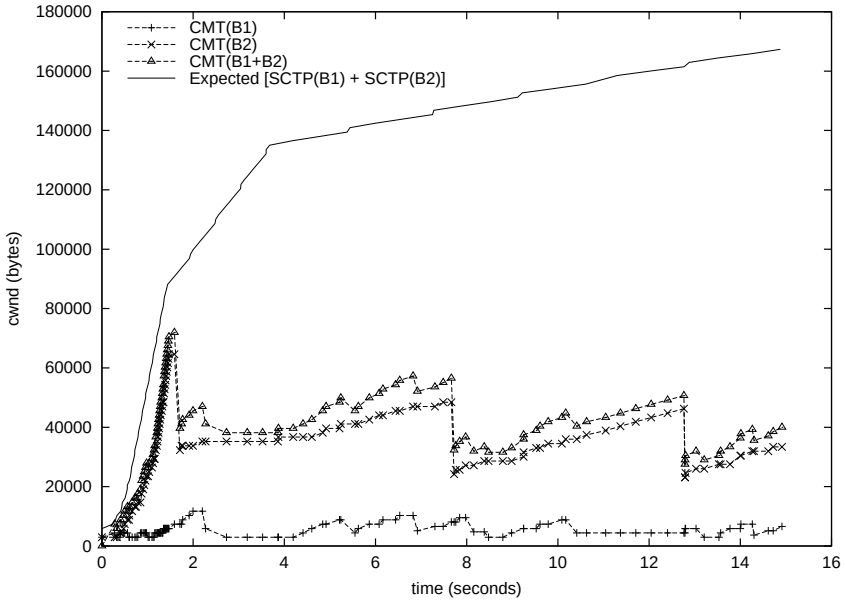


Figure 2.2: CMT with SCTP: Evolution of the different cwnds

the several cwnd reductions for both B_1 and B_2, for instance, the cwnd for B_2 is cut in half at times 1.8 sec., 7.8 sec., and 12.7 sec. Normally cwnd reductions are seen when a sender detects loss, but for Figure 2.2, no packet loss was simulated! The aggregate cwnd evolution ($\triangle$) is significantly below the expected aggregate cwnd evolution (–).

We identify and resolve three negative side-effects of reordering introduced by CMT that must be managed before the performance gains of parallel transfer can be fully achieved: (i) unnecessary fast retransmissions at the sender (Section 2.2), (ii) reduced cwnd growth due to fewer cwnd updates at the sender (Section 2.3), and (iii) increased ack traffic due to fewer delayed acks (Section 2.4).

A note on notation used in this chapter. CMT refers to a host performing concurrent multipath transfer using current SCTP. CMT$_s$, CMT$_c$, and CMT$_d$ refer to a host performing CMT with Split Fast Retransmit (SFR) (Section 2.2), Cwnd Update for CMT (CUC) (Section 2.3) and Delayed Ack for CMT (DAC) (Section 2.4) algorithms, respectively.

Multiple subscripts indicates use of more than one algorithm.

2.2 Preventing Unnecessary Fast Retransmissions - SFR Algorithm

When reordering is observed, a receiver sends gap reports (i.e., gap acks) to the sender which uses the reports to detect loss through a fast retransmission procedure similar to TCP's Fast Retransmit [8,65]. With CMT, unnecessary fast retransmissions can be caused due to reordering [40], with two negative consequences: (1) since each retransmission is assumed to occur due to a congestion loss, the sender reduces its cwnd for the destination on which the retransmitted data was outstanding, and (2) a cwnd overgrowth problem (discussed later in Chapter 6) causes a sender's cwnd to grow aggressively for the destination on which the retransmissions are sent, due to acks received for original transmissions. In Figure 2.2, each cwnd reduction observed for B_1 and B_2 is due to an unnecessary fast retransmission. These unnecessary retransmissions are due to sender-introduced reordering, and are not spurious retransmissions due to network effects [50, 52].

Conventional interpretation of a SACK chunk in SCTP (or SACK options in TCP) is that gap reports imply possible loss. The probability that a TSN is lost, as opposed to being reordered, increases with the number of gap reports received for that TSN. Due to sender-induced reordering, a CMT sender needs additional information to infer loss. Gap reports *alone* do not (necessarily) imply loss; but a sender can infer loss using gap reports *and* knowledge of each TSN's destination.

Algorithm Details: The proposed solution to address the side-effect of incorrect cwnd evolution due to unnecessary fast retransmissions is the *Split Fast Retransmit (SFR)* algorithm (Figure 2.3). SFR extends a previous incarnation which could not handle *cycling changeover* [40]. SFR introduces a *virtual queue* per destination within the sender's retransmission queue. A sender deduces missing reports for a TSN using SACK information in conjunction with state maintained about the transmission destination for each TSN

in the retransmission queue. With SFR, a multihomed sender correctly applies the fast retransmission procedure on a *per destination* basis. An advantage of SFR is that only the sender's behavior is affected; the SCTP receiver is unchanged.

SFR introduces two additional variables per destination at a sender:

1. *highest_in_sack_for_dest* - stores the highest TSN acked per destination by the SACK being processed.

2. *saw_newack* - a flag used during the processing of a SACK to infer the causative TSN(s)'s destination(s). Causative TSNs for a SACK are those TSNs which caused the SACK to be sent (i.e., TSNs that are acked in this SACK, and are acked for the first time).

On receipt of a SACK containing gap reports [Sender side behavior]:

1) $\forall$ destination addresses d_i, initialize $d_i.saw\_newack$ = FALSE;
2) **for** each TSN t_a being acked that has not been acked
   in any SACK thus far **do**
   let d_a be the destination to which t_a was sent;
   set $d_a.saw\_newack$ = TRUE;
3) $\forall$ destinations d_n, set $d_n.highest\_in\_sack\_for\_dest$ to highest TSN
   being newly acked on d_n;
4) to determine whether missing report count for a TSN t_m should be incremented:
   let d_m be the destination to which t_m was sent;
   if ($d_m.saw\_newack$ == TRUE) **and**
   ($d_m.highest\_in\_sack\_for\_dest > t_m$) **then**
   increment missing report count for t_m;
   else do not increment missing report count for t_m;

Figure 2.3: Split Fast Retransmit (SFR) Algorithm – Eliminating unnecessary fast retransmissions

In Figure 2.3, step (2) sets *saw_newack* to TRUE for the destinations to which the newly acked TSNs were sent. Step (3) tracks, on a per destination basis, the highest TSN being acked. Step (4) uses information gathered in steps (2) and (3) to infer missing TSNs. Two conditions in step (4) ensure correct missing reports: (a) TSNs to be marked should be outstanding on the same destination(s) as TSNs which have been newly acked, and (b) at least one TSN, sent later than the missing TSN, but *to the same destination address*, should be newly acked.

2.3 Avoiding Reduction in Cwnd Updates - CUC Algorithm

The cwnd evolution algorithm for SCTP [65] (and analogously for SACK TCP [8, 26]) allows growth in cwnd only when a new cum ack is received by a sender. When SACKs with unchanged cum acks are generated (say due to reordering) and later arrive at a sender, the sender does not modify its cwnd. This mechanism again reflects the conventional view that a SACK which does not advance the cum ack indicates possibility of loss due to congestion.

Since a CMT receiver naturally observes reordering, many SACKs are sent containing new gap reports but not new cum acks. When these gaps are later acked by a new cum ack, cwnd growth occurs, but only for the data newly acked in the most recent SACK. Data previously acked through gap reports will not contribute to cwnd growth. This behavior prevents sudden increases in the cwnd resulting in bursts of data being sent. Even though data may have reached the receiver "in-order per destination," without changing the current handling of cwnd, the updated cwnd will not reflect this fact.

This inefficiency can be attributed to the current design principle that the cum ack in a SACK, which tracks the latest TSN received in-order at the receiver, applies to an entire association, not per destination. TCP and current SCTP (i.e., SCTP without CMT) use only one destination address at any given time to transmit new data to, and hence,

this design principle works fine. Since CMT uses multiple destinations simultaneously, cwnd growth in CMT demands tracking the latest TSN received in-order *per destination*, information not coded directly in a SACK.

We propose a cwnd growth algorithm to track the earliest outstanding TSN *per destination* and update the cwnd, even in the absence of new cum acks. The proposed *Cwnd Update for CMT (CUC)* algorithm uses SACKs and knowledge of transmission destination for each TSN to deduce in-order delivery per destination. The crux of the CUC algorithm is to track the earliest outstanding data *per destination*, and use SACKs which ack this data to update the corresponding cwnd. In understanding our proposed solution, we remind the reader that gap reports alone do not (necessarily) imply congestion loss; SACK information is treated only as a concise description of the TSNs received by the receiver.

Algorithm Details: Figure 2.4 shows the proposed CUC algorithm. A *pseudo-cumack* tracks the earliest outstanding TSN per destination at the sender. An advance in a pseudo-cumack triggers a cwnd update for the corresponding destination, even when the actual cum ack is not advanced. The pseudo-cumack is used for cwnd updates; only the actual cum ack can dequeue data in the sender's retransmission queue since a receiver can renege on data that is not cumulatively acked. An advantage of CUC is that only the sender's behavior is affected; the SCTP receiver is unchanged.

The CUC algorithm introduces three variables per destination at a sender:

1. *pseudo_cumack* - maintains earliest outstanding TSN.
2. *new_pseudo_cumack* - flag to indicate if a new pseudo-cumack has been received.
3. *find_pseudo_cumack* - flag to trigger a search for a new pseudo-cumack. This flag is set after a new pseudo-cumack has been received.

At beginning of an association [Sender side behavior]:

∀ destinations d, reset

d.find_pseudo_cumack = TRUE;

On receipt of a SACK [Sender side behavior]:

1) ∀ destinations d, reset *d.new_pseudo_cumack* = FALSE;
2) **if** the SACK carries a new cum ack **then**
   for each TSN t_c being cum acked for the first time, that was not acked through prior gap reports **do**
   (i) let d_c be the destination to which t_c was sent;
   (ii) set $d_c.find\_pseudo\_cumack$ = TRUE;
   (iii) set $d_c.new\_pseudo\_cumack$ = TRUE;
3) **if** gap reports are present in the SACK **then**
   for each TSN t_p being processed from the retransmission queue **do**
   (i) let d_p be the destination to which t_p was sent;
   (ii) **if** ($d_p.find\_pseudo\_cumack$ == TRUE) **and**
   t_p was not acked in the past **then**
   $d_p.pseudo\_cumack = t_p$;
   $d_p.find\_pseudo\_cumack$ = FALSE;
   (iii) **if** t_p is acked via gap reports for first time **and**
   ($d_p.pseudo\_cumack == t_p$) **then**
   $d_p.new\_pseudo\_cumack$ = TRUE;
   $d_p.find\_pseudo\_cumack$ = TRUE;
4) **for** each destination d **do**
   if (*d.new_pseudo_cumack* == TRUE) **then** update cwnd as per [63, 65];

Figure 2.4: Cwnd Update for CMT (CUC) Algorithm – Handling side-effect of reduced cwnd growth due to fewer cwnd updates

In Figure 2.4, step (2) initiates a search for a new *pseudo_cumack* by setting *find_pseudo_cumack* to TRUE for the destinations on which TSNs newly acked were outstanding. A cwnd update is also triggered by setting *new_pseudo_cumack* to TRUE for those destinations. Step (3) then processes the outstanding TSNs at a sender, and tracks on a per destination basis, the TSN expected to be the next *pseudo_cumack*. Step (4) finally updates the cwnd for a destination if a new *pseudo_cumack* was seen for that destination.

2.4 Curbing Increase in Ack Traffic - DAC Algorithm

Sending an ack after receiving every two data PDUs (i.e., delayed acks) in SCTP (and TCP) reduces ack traffic in the Internet, thereby saving processing and storage at routers on the ack path. SCTP specifies that a receiver should use the delayed ack algorithm as given in RFC 2581 [8], where acks are delayed only as long as the receiver receives data in order. Reordered PDUs should be acked immediately. With CMT's frequent reordering, this rule causes an SCTP receiver to frequently *not* delay acks. Hence a negative side-effect of reordering with CMT is increased ack traffic.

To prevent this increase, we propose that a CMT receiver ignore the rule mentioned above. That is, a CMT receiver does not immediately ack an out-of-order PDU, but delays the ack. Thus, a CMT receiver always delays acks, irrespective of whether or not data is received in order[1]. Though this modification eliminates the increase in ack traffic, RFC 2581's rule has another purpose which gets hampered.

An underlying assumption that pervades SCTP's (and TCP's) design is that data in general arrives in order; data received out-of-order indicates possible loss. According to RFC 2581, a receiver should immediately ack data received above a gap in the sequence space to accelerate loss recovery with the fast retransmit algorithm [8]. In SCTP, four acks with

[1] We do not modify a receiver's behavior when an ack being delayed can be piggy-backed on reverse path data, or when the delayed ack timer expires.

missing reports for a TSN indicate that a receiver received at least four data PDUs sent after the missing TSN. Receipt of four missing reports for a TSN triggers the sender's fast retransmit algorithm. In other words, the sender has a *reordering threshold* (or *dupack threshold*) of four PDUs. Since a CMT receiver cannot distinguish between loss and reordering introduced by a CMT sender, the modification suggested above by itself would cause the receiver to delay acks even in the face of loss. When a loss does occur with our modification to a receiver, fast retransmit would be triggered by a CMT sender only after the receiver receives eight(!) data PDUs sent after a lost TSN - an overly conservative behavior.

The effective increase in reordering threshold at a sender can be countered by reducing the actual number of acks required to trigger a fast retransmit at the sender, i.e., by increasing the number of missing reports registered per ack. In other words, if a sender can increment the number of missing reports more accurately per ack received, fewer acks will be required to trigger a fast retransmit. A receiver can provide more information in each ack to assist the sender in accurately inferring the number of missing reports per ack for a lost TSN. We propose that in each ack, a receiver report the number of data PDUs received since the previous ack was sent. A sender then infers the number of missing reports per TSN based on the TSNs being acked in a SACK, number of PDUs reported by the receiver, and knowledge of transmission destination for each TSN. We note that additionally, heuristics (as proposed in [13]) may be used at a CMT sender to address network induced reordering.

Algorithm Details: The proposed *Delayed Ack for CMT (DAC)* algorithm (Figure 2.5) specifies a receiver's behavior on receipt of data, and also a sender's behavior when the missing report count for a TSN needs to be incremented. Since SCTP (and TCP) acks are cumulative, loss of an ack will result in loss of the data PDU count reported by the receiver, but the TSNs will be acked by the following ack. Receipt of this following

On receipt of a data PDU [Receiver side behavior]:

1) delay sending ack as given in [65], with the additional rule that the ack should be delayed even if reordering is observed.
2) in ack, report number of data PDUs received since sending of previous ack.

When incrementing missing report count through SFR:Step (4)

4) to determine whether missing report count for a TSN t_m should be incremented: let d_m be the destination to which t_m was sent;
   if ($d_m.saw\_newack$ = TRUE) **and** ($d_m.highest\_in\_sack\_for\_dest > t_m$) **then**
   (i) **if** ($\forall$ destinations d_o where $d_o \neq d_m$, $d_o.saw\_newack$ == FALSE) **then**
   (all newly acked TSNs were sent to the same destination as t_m)
   (a) **if** ($\exists$ newly acked TSNs t_a, t_b such that $t_a < t_m < t_b$) **then**
   (conservatively) increment missing report count for t_m by 1;
   (b) **else if** ($\forall$ newly acked TSNs t_a, $t_a > t_m$) **then**
   increment missing report count for t_m by number of PDUs reported by receiver;
   (ii) **else**
   (Mixed SACK - newly acked TSNs were sent to multiple destinations)
   (conservatively) increment missing report count for t_m by 1;

Figure 2.5: Delayed Ack for CMT (DAC) Algorithm – Handling side-effect of increased ack traffic

ack can cause ambiguity in inferring missing report count per destination. Our algorithm conservatively assumes a single missing report count per destination in such ambiguous cases. The DAC algorithm requires modifications to both the sender and the receiver.

No new variables are introduced in DAC, as we build on the SFR algorithm. An additional number is reported in the SACKs for which we propose using the first bit of the flags field in the SACK chunk header - 0 indicates a count of one PDU (default SCTP behavior), and 1 indicates two PDUs.

In Figure 2.5, at the receiver side, steps (1) and (2) are self explanatory. The sender side algorithm modifies step (4) of SFR, which determines whether the missing report count

should be incremented for a TSN. DAC dictates *how many* to increment by. Step (4-i) checks if only one destination was newly acked, and allows incrementing missing reports by more than one for TSNs outstanding to that destination. Further, all newly acked TSNs should have been sent later than the missing TSN. If there are newly acked TSNs that were sent before the missing TSN, step (4-i-a) conservatively increments missing reports by only one. If more than one destinations are newly acked, step (4-ii) conservatively increments by only one.

Figure 2.6 shows cwnd evolution for CMT after including the SFR, CUC and DAC algorithms, i.e., CMT$_{scd}$. With the negative side-effects addressed, we expected to see CMT$_{scd}$'s cwnd growth to come close to the expected aggregate cwnd growth. In fact, we observed that CMT$_{scd}$ cwnd growth *exceeded* the expected aggregate cwnd growth!

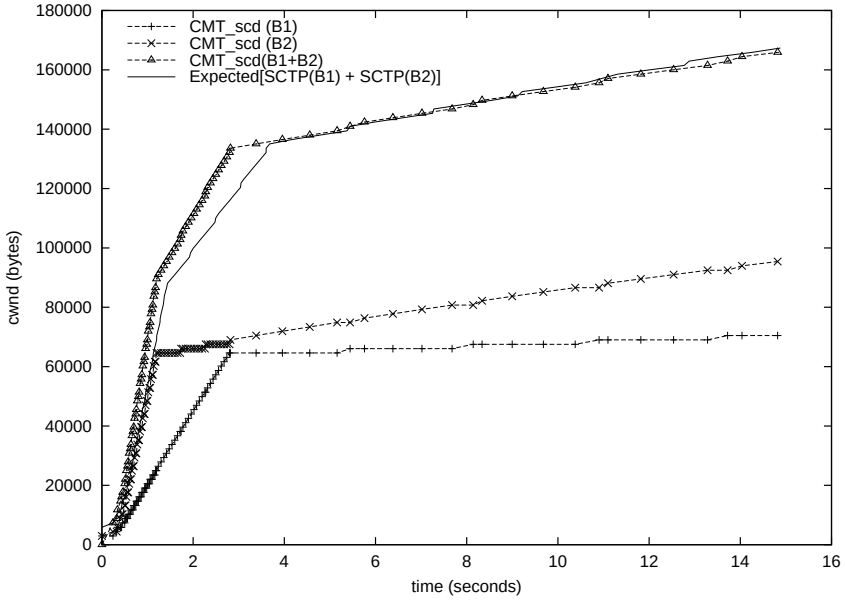


Figure 2.6: CMT$_{scd}$: Evolution of the different cwnds

To explain this surprising result, we remind the reader that the expected aggregate cwnd is the sum of the cwnd growth of two independent SCTP runs, each using one of the two destination addresses as its primary destination. In each SCTP run, one delayed ack can

increase the cwnd by at most one MSS during slow start, even if the ack acks more than one MSS worth of data. On the other hand, we observe with CMT$_{scd}$ that if a delayed ack simultaneously acks an MSS of data on each of the two destinations, the sender can simultaneously increase each of two cwnds by one MSS. Thus, a single delayed ack in CMT$_{scd}$ that acks data flows on two paths causes an aggregate cwnd growth of two MSS. With delayed acks during slow start, each SCTP association grows its cwnd by 1.5 times each RTT, whereas CMT$_{scd}$ can increase its cwnd by more than 1.5 times in each RTT (up to two times in the best case where every delayed ack acks an MSS on each path). *Delayed acks which simultaneously contribute to the cwnd growth of two destinations helped the aggregate cwnd growth of CMT$_{scd}$ exceed expected aggregate cwnd growth.*

This phenomenon occurs in slow start, therefore benefiting CMT$_{scd}$ initially and during some timeout recovery periods. Though the aggregate cwnd growth exceeds expected aggregate cwnd growth, we argue that the sender is not overly aggressive, i.e., not TCP-unfriendly. The sender is able to clock out more data due to delayed acks that ack data flows on multiple paths. The sender does not create bursts of data during slow start, and builds up the ack clock as expected. Though it does not improve CMT$_{scd}$'s performance significantly, *this phenomenon demonstrates a benefit of* sequence space sharing *among flows on different paths that occurs within a CMT$_{scd}$ association.*[2]

[2] Henceforth, we will refer to CMT$_{scd}$ as simply CMT.

Chapter 3

RETRANSMISSION POLICIES FOR CMT

In this chapter, we explore several retransmission policies for CMT (Section 3.1). We motivate and present two modifications to CMT mechanisms that are needed to allow redirecting retransmissions to a different destination than the original (Section 3.3). We then evaluate CMT's performance using the different retransmission policies against *App-Stripe*, an idealized application that stripes data across multiple paths (Sections 3.2 and 3.4).

3.1 CMT Retransmission Policies

Multiple paths present an SCTP sender with several options where to send a retransmission. But the choice is not well-informed since SCTP restricts sending new data, which can act as probes for information (such as available bandwidth, loss rate and RTT), to only one primary destination. Consequently, an SCTP sender has minimal information about paths to a receiver other than the path to the primary destination.

On the other hand, a CMT sender maintains accurate information about all paths, since new data is being sent to all destinations concurrently. This information allows a CMT sender to better decide where to retransmit. That is, a CMT sender can choose the retransmission destination for each loss recovery independently using the most recent information for its decision.

We present five retransmission policies for CMT [36]. In four policies, a retransmission may be sent to a destination other than the one used for the original transmission. Previous research on SCTP retransmission policies shows that sending retransmissions to an alternate destination degrades performance primarily because of the lack of sufficient traffic on alternate paths [19]. With CMT, data is concurrently sent on all paths, thus the results in [19] are not applicable. The five retransmission policies for CMT are:

- **RTX-SAME** - Once a new data chunk is scheduled and sent to a destination, all retransmissions of that chunk are sent to the same destination (until the destination is deemed *inactive* due to failure [65]).
- **RTX-ASAP** - A retransmission of a data chunk is sent to any destination for which the sender has cwnd space available at the time of retransmission. If multiple destinations have available cwnd space, one is chosen randomly.
- **RTX-CWND** - A retransmission is sent to the destination for which the sender has the largest cwnd. A tie is broken by random selection.
- **RTX-SSTHRESH** - A retransmission is sent to the destination for which the sender has the largest ssthresh. A tie is broken by random selection.
- **RTX-LOSSRATE** - A retransmission is sent to the destination with the lowest loss rate path. If multiple destinations have the same loss rate, one is selected randomly.

Of the policies, RTX-SAME is simplest. RTX-ASAP is a “hot-potato” policy - retransmit as soon as possible without regard to loss rate. RTX-CWND and RTX-SSTHRESH practically track, and attempt to move retransmissions onto the path with the estimated lowest loss rate. Since ssthresh is a slower moving variable than cwnd, the values of ssthresh may better reflect the conditions of the respective paths. RTX-LOSSRATE uses information about loss rate provided by an “oracle” - information that RTX-CWND and

RTX-SSTHRESH estimate. This policy represents a hypothetically ideal case; hypothetical since in practice, a sender typically does not know *a priori* path loss rates; ideal since the path with the lowest loss rate has highest chance of having a packet delivered. We hypothesized that retransmission policies that take loss rate into account would outperform ones that do not.

We do not propose different policies for new transmissions - we assume that a sender does not know path properties *a priori* and can therefore only react to network events such as congestion losses (see Chapter 2, Section 2.1 for our transmission policy). This assumption holds true particularly in the Internet where the path properties are changing.

3.2 Evaluation Methodology

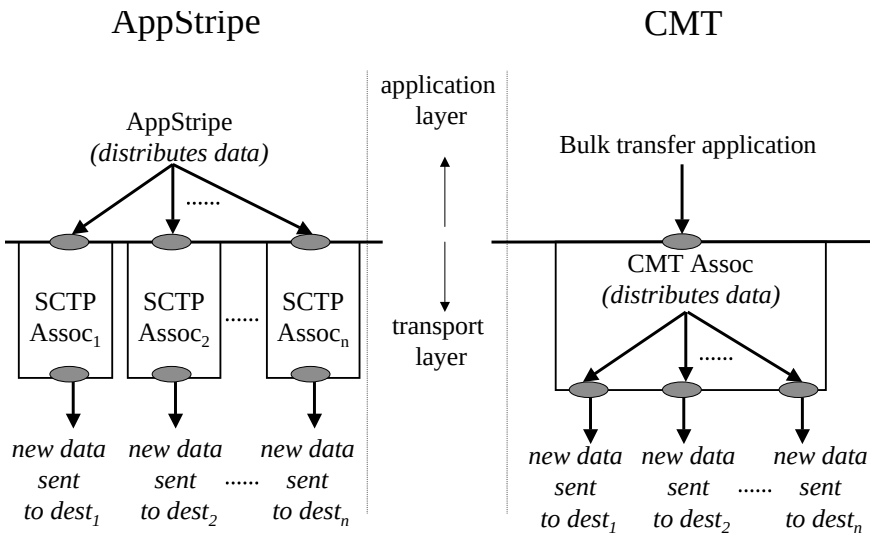


Figure 3.1: Schematic - AppStripe and CMT

As a reference, we use *AppStripe* - a hypothetical multihome-aware application that achieves the highest throughput possible by an application that distributes data across multiple SCTP associations (see Figure 3.1). We emphasize that AppStripe performs

idealized scheduling at the application layer, and is not doable in practice. AppStripe performs end-to-end load sharing at the application layer; CMT performs it at the transport layer.

We simulate AppStripe by post-processing simulation traces. We simulate separate file transfers over multiple separate SCTP associations, each on a separate path to the receiver. These SCTP associations use the *Retransmit to Same Destination* policy, where all retransmissions are sent to the same destination as the original transmission, as recommended in [17]. To find an "optimal" transfer time, we use these multiple traces to extract the time when the total amount of data transferred, across the multiple associations, equals the desired transfer size.

AppStripe hypothetically assumes the ability of an application to schedule data to each transport association immediately when a transport association is able to send data to a receiver. Such an ability requires a complex application-transport interface, which to our knowledge, is not realized in practice today. A typical application distributing data over multiple associations would have to use a heuristic to decide the fraction of data to be scheduled on each association. Thus AppStripe's performance in our experiments represents better achievable separation of data over multiple paths than is doable in practice.

The simulation topology (see Figure 3.2) is simple - edge links represent the last hop, and core links represent end-to-end conditions on the Internet. This simulation topology does not account for some effects seen in the Internet and other real networks such as network induced reordering, delay spikes, etc.; these effects are beyond the scope of this study. Our simulation evaluation provides insight into the fundamental differences between AppStripe and CMT, and between the different retransmission policies in a constrained environment. We chose a simple topology to avoid influence of other effects, and to focus on performance differences which we believe should hold true in a real environment as well. The loss rate on Path 1 is maintained at 1%, and on Path 2 is varied from

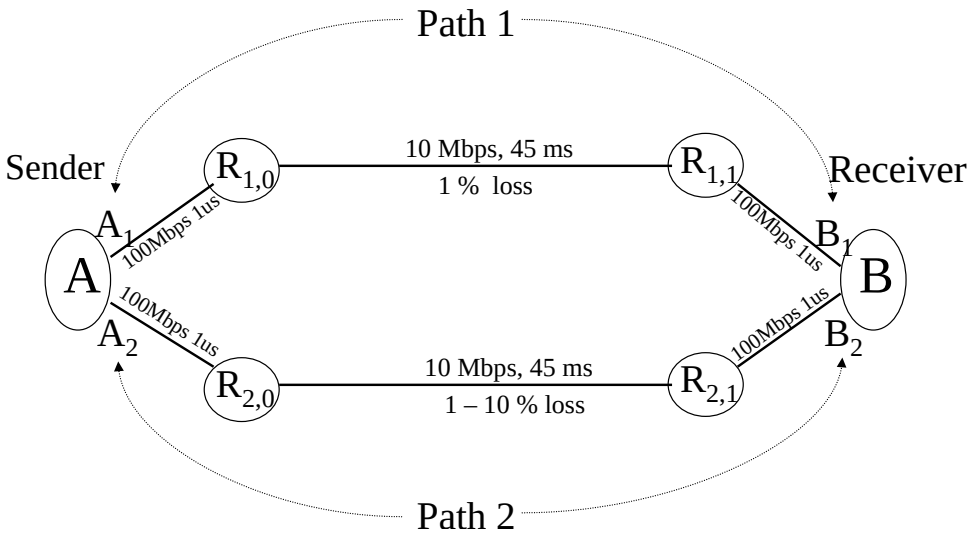


Figure 3.2: Simulation topology used for evaluation

1 to 10%. A loss rate of 1% means a forward path loss rate of 1%, and a reverse path loss rate of 1%. The loss events are independent per packet (i.e., we use a Bernoulli loss model).

Our choice of simulation parameters was based on our understanding that end-to-end throughput is most influenced by loss rate and delay. We focus on loss rate differences since we believe loss rate has a more significant impact on the retransmission policy. The influence of different delays and delay combinations on CMT is studied in Chapter 4.

The bandwidths were chosen to be high enough so that end-to-end delays are dominated by propagation delay. The relative bandwidths of the links were chosen so that any queuing happens at intermediate routers where packets are dropped with an independent and fixed loss probability. End-to-end delay was chosen as 45ms to represent a typical US coast-to-coast delay.

3.3 Modifications to Protocol Mechanisms

Two modifications are needed to allow redirecting retransmissions to a different destination than the original.

3.3.1 CUCv2: Modified CUC Algorithm

The CUC algorithm (Figure 2.4) enables correct cwnd updates in the face of increased reordering due to CMT. To recap Section 2.3, the CUC algorithm recognizes a set of TSNs outstanding per destination, and the per-destination *pseudo_cumack* traces the left edge of this list of TSNs, per destination. CUC assumes that retransmissions are sent to the same destination as the original transmission. The per-destination *pseudo_cumack* therefore moves whenever the corresponding left edge is acked; the TSN on the left edge being acked may or may not have been retransmitted.

If the assumption about the retransmission destination is violated, and a retransmission is made to a different destination from the original, CUC cannot faithfully track the left edge on either destination. We modify CUC to permit the different retransmission policies. The modified algorithm, named CUCv2 is shown in Figure 3.3.

CUCv2 recognizes that a distinction can be made about the TSNs outstanding on a destination - those that have been retransmitted, and those that have not. CUCv2 maintains two left edges for these two sets of TSNs - *rtx_pseudo_cumack* and *pseudo_cumack*. Whenever either of the left edges moves, a cwnd update is triggered.

3.3.2 Spurious Timeout Retransmissions

When a timeout occurs, an SCTP sender is expected to bundle and send as many of the earliest TSNs outstanding on the destination for which the timeout occurred as can fit in an MSS (Maximum Segment Size) PDU. Per RFC 2960, more TSNs that are outstanding

At beginning of an association [Sender side behavior]:

∀ destinations d, reset

d.find_pseudo_cumack = *d.find_rtx_pseudo_cumack* = TRUE;

On receipt of a SACK [Sender side behavior]:

1) ∀ destinations d, reset

d.new_pseudo_cumack = *d.new_rtx_pseudo_cumack* = FALSE;

2) **if** the ack carries a new cum ack **then**

for each TSN t_c being cum acked for the first time, that was not acked through prior gap reports **do**

(i) let d_c be the destination to which t_c was sent;

(ii) set $d_c.find\_pseudo\_cumack$, $d_c.find\_rtx\_pseudo\_cumack$, $d_c.new\_pseudo\_cumack$, $d_c.new\_rtx\_pseudo\_cumack$ to TRUE;

3) **if** gap reports are present in the ack **then**

for each TSN t_p being processed from the retransmission queue **do**

(i) let d_p be the destination to which t_p was sent;

(ii) **if** ($d_p.find\_pseudo\_cumack$ == TRUE)

and t_p was not acked in the past

and t_p was not retransmitted **then**

$d_p.pseudo\_cumack$ = t_p;

$d_p.find\_pseudo\_cumack$ = FALSE;

(iii) **if** t_p is acked via gap reports for first time

and ($d_p.pseudo\_cumack$ == t_p) **then**

$d_p.new\_pseudo\_cumack$ = TRUE;

$d_p.find\_pseudo\_cumack$ = TRUE;

(iv) **if** ($d_p.find\_rtx\_pseudo\_cumack$ == TRUE)

and t_p was not acked in the past **and** t_p was retransmitted **then**

$d_p.rtx\_pseudo\_cumack$ = t_p;

$d_p.find\_rtx\_pseudo\_cumack$ = FALSE;

(v) **if** t_p is acked via gap reports for first time

and ($d_p.rtx\_pseudo\_cumack$ == t_p) **then**

$d_p.new\_rtx\_pseudo\_cumack$ = TRUE;

$d_p.find\_rtx\_pseudo\_cumack$ = TRUE;

4) **for** each destination d **do**

if (*d.new_pseudo_cumack* == TRUE)

or (*d.new_rtx_pseudo_cumack* == TRUE) **then**

update cwnd as per [63, 65];

Figure 3.3: CUCv2 Algorithm - Modified Cwnd Update for CMT (CUC) Algorithm

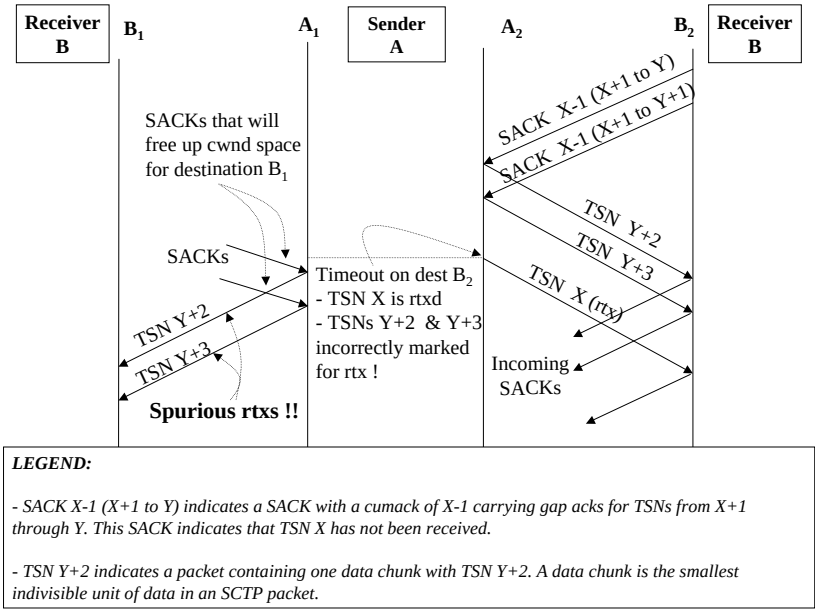


Figure 3.4: Example of spurious retransmissions after timeout in CMT

on that destination "should be marked for retransmission and sent as soon as cwnd allows (normally when a SACK arrives)". This rule is intuitive. While sending, retransmissions are generally given priority over new transmissions. As in TCP, the cwnd is also collapsed to 1 MSS for the destination on which a timeout occurs.

A timeout retransmission can occur in SCTP (as in TCP) for several reasons. One reason is loss of the fast retransmission of a TSN. Consider Figure 3.4. When a timeout occurs due to loss of a fast retransmission, some TSNs that were just sent to the destination on which the timeout occurred are likely awaiting acks (in Figure 3.4, TSNs Y+2 and Y+3). These TSNs get incorrectly marked for retransmission on timeout. With the different CMT retransmission policies, these retransmissions may be sent to a different destination than the original transmission. In Figure 3.4, spurious retransmissions of TSNs Y+2 and Y+3 are sent to destination B_1, on receipt of acks freeing up cwnd space for destination B_1. Spurious retransmissions are exacerbated in CMT, as shown through this illustration, due to the possibility of sending data (including retransmissions) to multiple destinations concurrently.

We simulated the occurrence of such spurious retransmissions with the different retransmission policies in CMT. The simulation topology used was the one described in Section 3.2. Figure 3.5 shows the ratio of retransmissions relative to the number of actual packet drops at the router. Ideally, the two numbers should be equal; all curves should be straight lines at $y = 1$. Figure 3.5 shows that spurious retransmissions occur commonly in CMT with the different retransmission policies.

We propose a heuristic to avoid these spurious retransmissions. Our heuristic assumes that a timeout cannot be triggered on a TSN until the TSN has been outstanding for at least one RTT. Thus, if a timeout is triggered, TSNs which were sent within one RTT are not marked for retransmission. We use an average measure of the RTT for this purpose - the smoothed RTT, which is maintained at a sender. This heuristic requires the sender

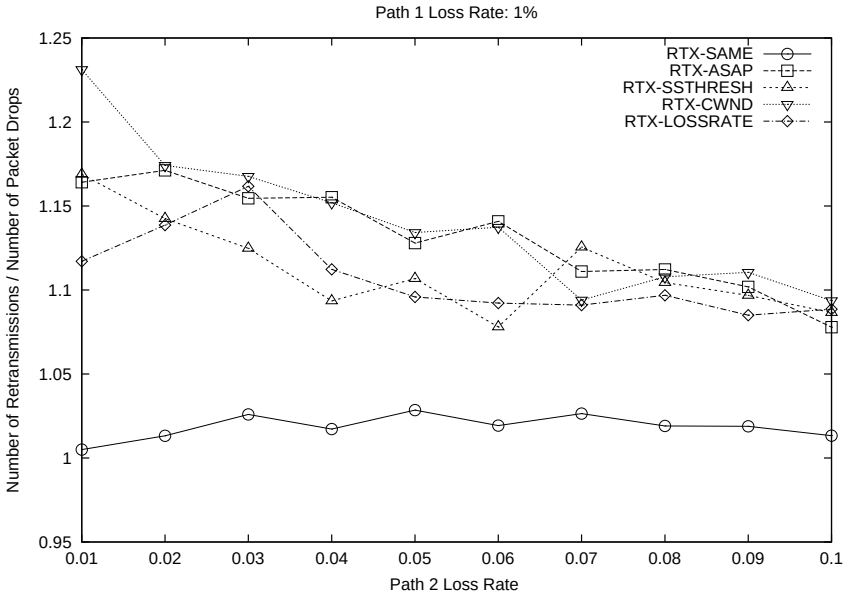


Figure 3.5: Spurious retransmissions in CMT without RTT heuristic

to maintain a timestamp for each TSN indicating the time at which the TSN was last transmitted (or retransmitted). Figure 3.6 shows how the application of this heuristic dramatically reduces spurious retransmissions.

3.4 Evaluation of CMT vs. AppStripe

Figure 3.7(a) compares the time taken to transfer an 8MB file using CMT with the five retransmission policies, vs. using AppStripe. The x-axis represents different loss rates on Path 2. Each plotted value is the mean of at least 30 simulation runs. Overall, AppStripe ($\times$ in Figure 3.7(a)) performs worst. That is, CMT using any of the retransmission policies performs better than AppStripe; some policies better than others. At a 7% loss rate on Path 2, AppStripe takes 40.4 seconds to transfer an 8 MB file, whereas CMT using RTX-SAME or RTX-CWND takes roughly 35 or 33 seconds, respectively. We first discuss the performance difference between CMT in general and AppStripe.

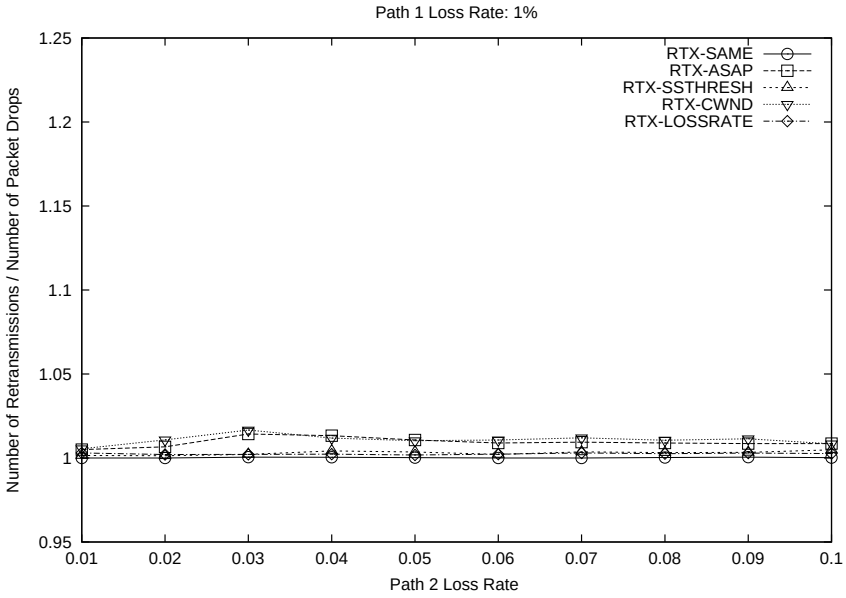
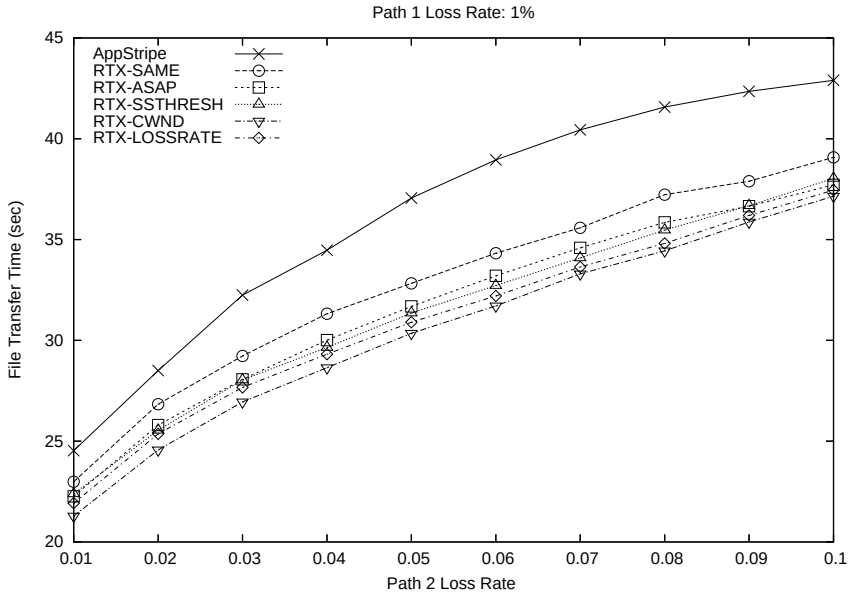


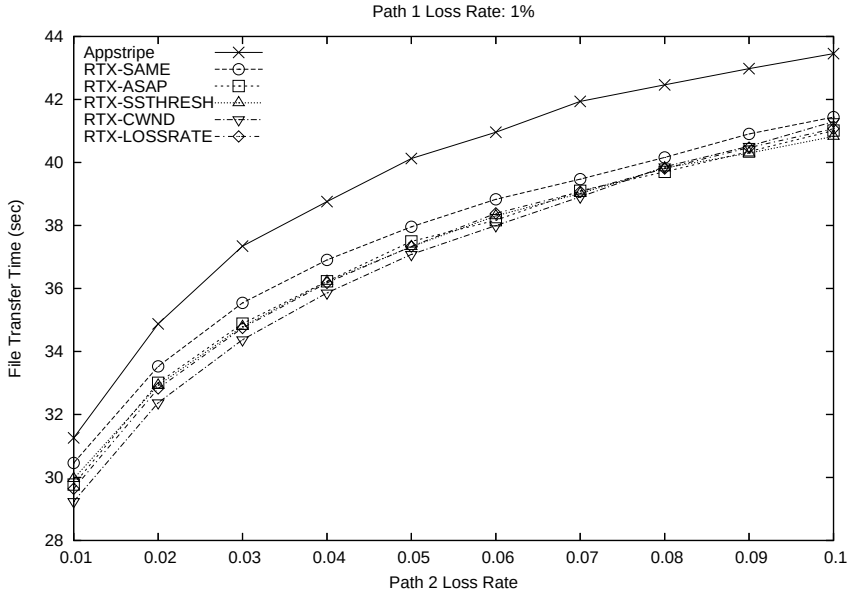
Figure 3.6: Spurious retransmissions in CMT with RTT heuristic

CMT using any retransmission policy performs better than AppStripe, particularly as the loss rate on Path 2 increases. Note that our AppStripe represents *the best possible performance* expected by an application that stripes data over multiple SCTP associations. AppStripe is an idealized case; CMT's performance gain over a practical AppStripe implementation would be even larger since a practical implementation has to optimally stripe data across paths that have different and changing delays and loss rates. Such striping may require information from the transport layer (such as current cwnd and RTT), that may not be readily available to the application.

CMT performs better than AppStripe for two reasons. First, and significant, CMT is more resilient to reverse path loss than AppStripe. CMT uses a single sequence space (TSN space, used for congestion control and loss detection and recovery) across an association's multiple paths, whereas AppStripe by design uses an independent sequence space per path. Since acks are cumulative, sharing of sequence spaces across paths helps



(a)



(b)

Figure 3.7: Path 1 loss rate = 1%, performance of AppStripe vs. CMT with different policies, under (a) equal path delays (Path 1 = 45ms, Path 2 = 45ms), and (b) unequal path delays (Path 1 = 45 ms, Path 2 = 90 ms)

a CMT sender receive ack info on either of the return paths. Thus, CMT effectively uses *both* return paths for communicating ack info to the sender, whereas each association in AppStripe cannot help the other "ack-wise". These results demonstrate the significant result that CMT's sharing of sequence space across paths is *an inherent benefit that performing load sharing at the transport layer has over performing it at the application layer*.

We emphasize that ack loss can cause throughput degradation, especially at higher loss rates. Ack loss can delay fast retransmissions by one or more RTTs, thus delaying cwnd increase. Increased ack loss can also increase the number of timeout retransmissions when the window is small (say during the initial part of an association, or after timeout recovery). These performance penalties accumulate over the lifetime of an association [36].

Second, CMT gets faster overall cwnd growth than AppStripe in slow start (see Section 2.4). As loss increases, the number of timeouts increases, and since slow start follows a timeout, the sender spends more time overall in slow start.

Extensive simulations show that unequal path delays do not impact the relative performance of AppStripe and CMT with the different policies. Figure 3.7(b) demonstrates this consistent behavior with unequal path delays of 45 ms on Path 1, and 90 ms on Path 2. Note that these results are consistent with Figure 3.7(a) which has equal delays of 45 ms on both paths.

While the performance differences between the retransmission policies in Figure 3.7 are negligible, these results use an 8MB receiver's buffer (rbuf) that does not constrain the sender—an unrealistic assumption which we will now drop as we further evaluate CMT in the following chapter.

Chapter 4

IMPLICATIONS OF A CONSTRAINED RECEIVE BUFFER

In Chapters 2 and 3, we operated under the strong and limiting assumptions that (1) the receive buffer (rbuf) was infinite, and (2) the bottleneck queues on the end-to-end paths used in CMT were independent of each other. In this chapter, we drop assumption (1) and investigate how a bounded rbuf affects CMT performance [34, 37] (assumption (2) is retained, and is discussed further in Chapter 7). While this chapter discusses performance considerations in the context of CMT, we note that these considerations apply to multipath transfer at other layers as well.

We present and describe the rbuf blocking problem (Section 4.1) and evaluate the different retransmission policies with different rbuf sizes to select a retransmission policy for CMT (Section 4.2). We then consider the impact of rbuf blocking with different loss rate and end-to-end delay combinations on the paths used for CMT (Section 4.3). We also verify a subset of our results in a realistic simulation topology with cross-traffic (Section 4.4). While SCTP supports unordered data delivery and multistreaming in an association [65], this chapter focuses on ordered data delivery over a single stream. A discussion on the insights gained through our evaluation of CMT concludes this chapter (Section 4.5).

The simulations in this chapter (except Section 4.4) use the same topology as described in Chapter 3 (Section 3.2). The topology is repeated in Figure 4.1 for convenience.

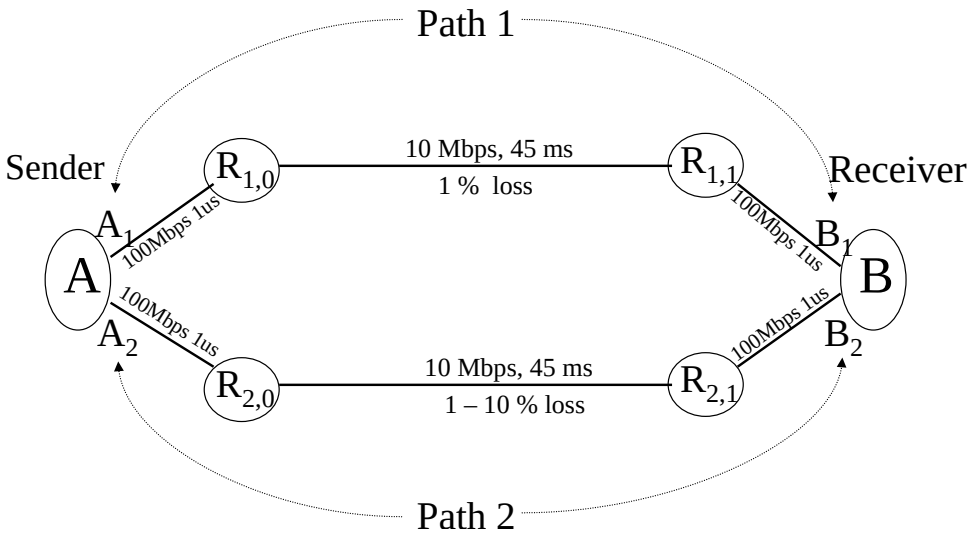


Figure 4.1: Simulation topology used for evaluation of retransmission policies

4.1 Receive Buffer Blocking in CMT: Problem Description

A transport layer receiver maintains rbuf space for incoming data for two reasons: (i) to handle out-of-order data, and (ii) to receive data at a rate higher than that of the receiving application's consumption. In SCTP (and TCP), a receiver advertises currently available rbuf space through window advertisements (normally accompanying acks) to a data sender. This value is the *advertised receive window (adv-rwnd)*. A sender computes a *peer-rwnd* to deduce how much *more* data can be buffered at the receiver. Beside the latest adv-rwnd received, the peer-rwnd takes into account data that has been sent but not yet acked by the receiver.

An SCTP receiver maintains a single rbuf across all *sub-association flows* in an association. We define a *sub-association flow* as the set of transport PDUs within an SCTP association that have the same destination address. For instance, in Figure 4.1, an SCTP

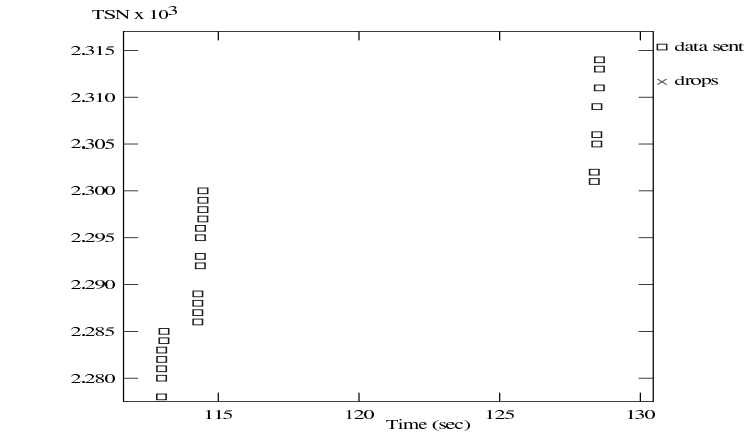
association from the sender to the receiver spanning the two paths will have two sub-association flows—one consisting of PDUs with destination B_1, and the other with destination B_2.

Consequent to a single rbuf at a receiver, an SCTP sender maintains a single peer-rwnd per association. Note that sender-side estimates such cwnd, ssthresh and RTT are maintained per destination—they represent the state of different network paths from a sender to each destination address. A sender has no reason to maintain separate rbufs or peer-rwnds per path since a receiver can consume data only in sequence, irrespective of the destination address they are sent to. An SCTP sender's sending rate is bound by both the peer-rwnd and the pertinent destination's cwnd, i.e., *min(peer-rwnd, cwnd)*.

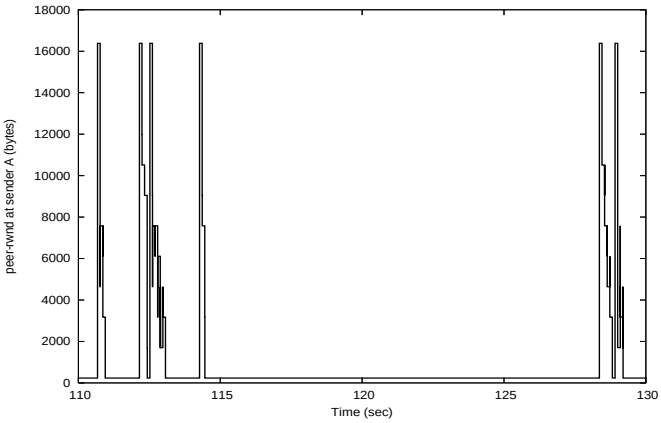
A CMT receiver maintains a single rbuf which is shared across all sub-association flows in an association[1]. Irrespective of the layer at which multipath transfer is performed, a similar shared buffer would exist at a receiver (likely at the transport or application layer). This buffer sharing degrades overall throughput. To help explain this degradation, Figure 4.2 shows an excerpt from a simulation of a CMT association using the topology shown in Figure 4.1. In this illustrative example, the rbuf is 16384 bytes (16KB), Path 1 (A_1 to B_1) has a loss rate of 1% and Path 2 (A_2 to B_2) has a loss rate of 10%, and RTX-SAME retransmission policy is used.

Figures 4.2(a) and (c) show TSN progression over Path 1 and Path 2, respectively, and Figure 4.2(b) shows peer-rwnd evolution at the sender (endpoint A) during the time interval from 110 to 130 seconds. Figure 4.2(a) shows that data transmission over the better (i.e., lower loss rate) path stops abruptly around 114.5 seconds and resumes around 128 seconds. This 13.5 second pause can be explained with the help of Figure 4.2(b). At

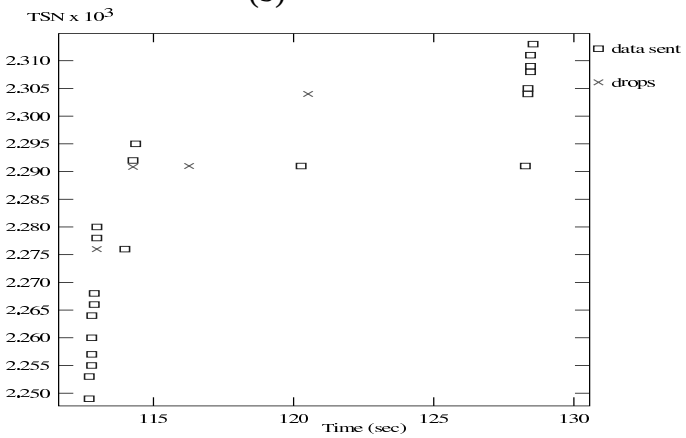
[1] SCTP supports *unordered* data delivery and *multistreaming* in an association [65], the impact of which we discuss further in Section 4.5. In the rest of this chapter, we assume ordered data delivery over a single stream.



(a)



(b)



(c)

Figure 4.2: Instantiation of rbuf blocking: (a) Progression of data sent to destination B_1 over Path 1 (loss rate 1%) over a select interval; (b) peer-rwnd value maintained at sender (endpoint A) over same interval; (c) Progression of data sent to destination B_2 over Path 2 (loss rate 10%) over same interval.

114.5 seconds, the peer-rwnd at the sender abruptly reduces from 16384 bytes to 236 bytes, constraining the sender from transmitting any new data. The cause for this abrupt rbuf reduction is explained as follows.

During the same time interval from 114.5 seconds to 128 seconds, Figure 4.2(c) shows that the lower quality (i.e., higher loss rate) path undergoes congestion, and recovers from losses through repeated retransmission timeouts - the longest recovery time being 8 seconds for TSN 2304. During this entire period of 13.5 seconds while loss recovery repeatedly occurs on Path 2, the receiver waits for retransmissions to come through, and is unable to deliver subsequent TSNs to the application (some of which were sent over Path 1). These subsequent TSNs are held in the transport layer rbuf until the retransmissions are received, thus blocking the rbuf and the peer-rwnd. Path 2 thus causes blocking of the rbuf, preventing data from being sent on either path and reducing overall throughput.

This example demonstrates how a shared rbuf causes a sub-association flow on a higher quality path to get lower throughput than expected. We note that the exact numbers used in this example do not hold special relevance. This example presents a phenomenon which occurs, in lesser or greater degree, throughout a CMT association.

Figure 4.3 shows the time taken to transfer an 8MB file using (i) CMT (with RTX-SAME retransmission policy) with a 16KB rbuf, and (ii) a single SCTP association which uses *only* the better path (Path 1 with loss rate 1%). Intuitively, using two paths should provide higher overall throughput than using one path. However, Figure 4.3 demonstrates that using two paths performs worse than using only the better path if a finite rbuf is shared across the paths[2]. This performance difference is due to rbuf blocking that occurs in CMT—an rbuf of 16KB does not constrain a single SCTP association (which uses one

[2] Presumably an SCTP sender does not have *a priori* knowledge about the better path and hence cannot always achieve best performance. We discuss *expected* SCTP performance later in Section 4.3.

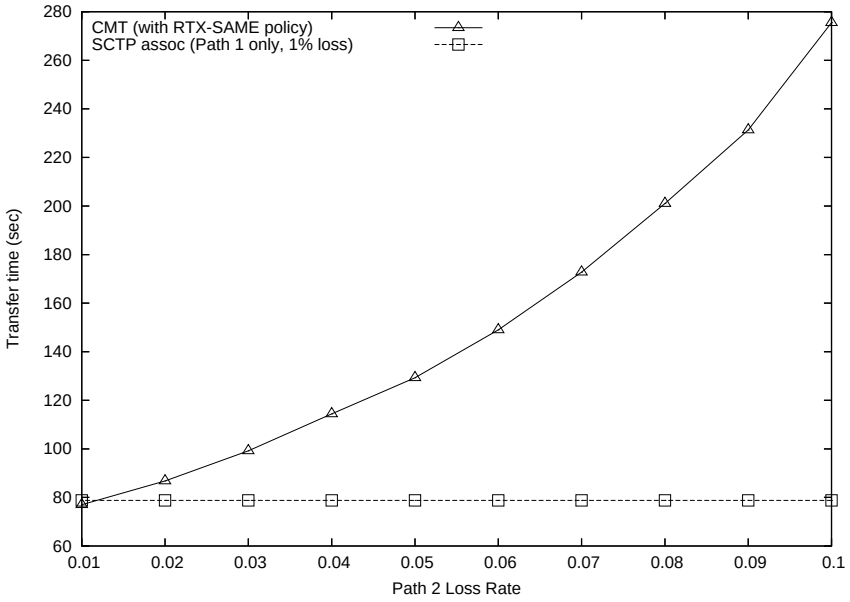


Figure 4.3: Rbuf blocking in CMT causes throughput degradation

lower loss rate path) as much as it constrains CMT (which uses two paths with different loss rates).

We emphasize that rbuf blocking is not unique to the transport layer; it applies to multi-path transfer at other layers as well. Rbuf blocking cannot be eliminated or avoided by moving CMT's parallelism to a different layer. For example, if the application layer distributes a single logical flow across multiple end-to-end paths, and the application layer receiver (the final destination) has finite buffer space, then rbuf blocking will occur.

After analyzing several traffic flows, we observe that chances of rbuf blocking are higher during periods of timeout recovery. A larger timeout recovery period due to back-to-back timeouts with exponential backoff results in an even higher probability that a finite rbuf blocks a sender. We therefore hypothesize that reducing (i) the number of timeouts, and/or (ii) the number of back-to-back timeout retransmissions will reduce the rbuf blocking problem. Consequently, we hypothesize that using a retransmission policy that will

reduce timeout periods will help reduce rbuf blocking.

4.2 Choosing a Retransmission Policy

We now evaluate our five retransmission policies for CMT (Section 3.1) operating under a constrained rbuf. Default rbuf values in commonly used operating systems today vary from 16KB to 64KB and beyond. We believe that today, when a desktop computer can have gigabytes of memory, having an rbuf of at least 64KB is reasonable. We first study and analyze performance of the different policies with an rbuf of 64KB in Section 4.2.1. This section provides insight into the causes of the performance differences between the retransmission policies. We then summarize performance of the different policies under more and less constraining rbufs varying from 16KB to 256KB in Section 4.2.2. This analysis provides an understanding of rbuf blocking impact on the different policies.

4.2.1 Evaluation with rbuf=64KB

Figure 4.4 shows the time taken for a CMT sender to transfer an 8MB file when the rbuf is set to 64KB, using the five retransmission policies. Each plotted value is the mean of at least 100 simulation runs. RTX-SAME, the simplest to implement, performs worst. Its performance gap with the other policies increases as the loss rate on Path 2 increases. RTX-ASAP performs better than RTX-SAME, but still considerably worse than the three loss-rate-based policies. We present two causes for these differences.

Cause 1: Figure 4.5 shows the number of retransmission timeouts experienced when using the different policies. One may conclude that improvement in using the loss-rate-based policies is due partly to fewer timeouts (and hence, timeout recovery periods). RTX-SAME does not consider loss rate and experiences the largest number of timeouts. RTX-ASAP does not consider loss rate and does better than RTX-SAME, but still experiences more timeouts than the loss-rate-based policies. This analysis supports our intuitive

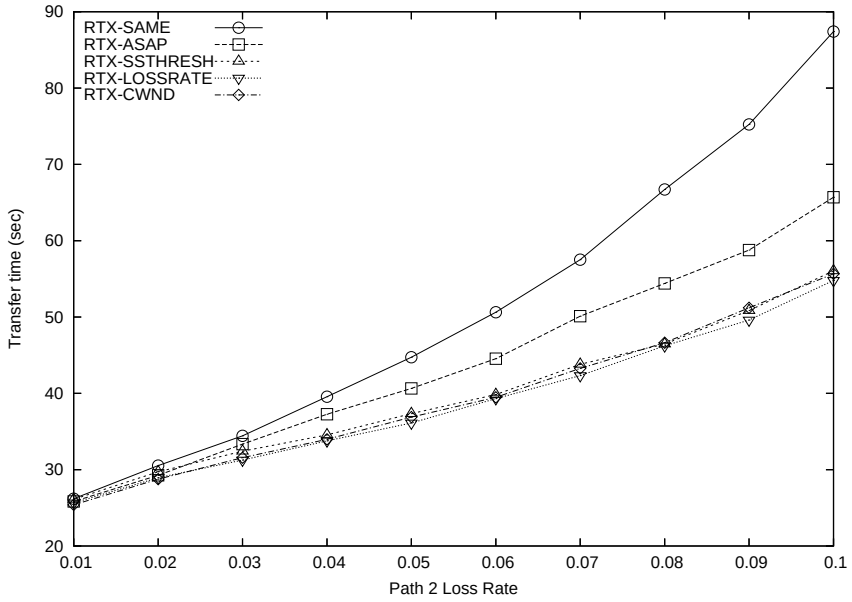


Figure 4.4: Time taken by CMT to transfer an 8MB file (rbuf: 64K, Path 1 loss rate: 1%)

hypothesis - taking path loss rate into consideration while deciding the retransmission destination improves the chances of a retransmission getting through, and improves overall performance due to reduction of rbuf blocking.

Cause 2: Figure 4.6(a) shows the average time taken to successfully communicate a TSN. This time is measured as the time taken from the first transmission of a TSN to the time when that TSN or one of its retransmissions finally reaches the receiver. RTX-SAME shows the highest average, suggesting that more transmissions may be needed for a successfully communicating a TSN. Since recovery via fast retransmission can happen only once for a given TSN, the number of consecutive timeouts may be higher with RTX-SAME than with the other policies. Each consecutive timeout causes a sender's retransmission timeout value to double, thus doubling the timeout recovery period. Recall that the longer the timeout recovery period, the higher the probability and longer the duration for rbuf blocking to occur. Thus, more consecutive timeouts will degrade performance.

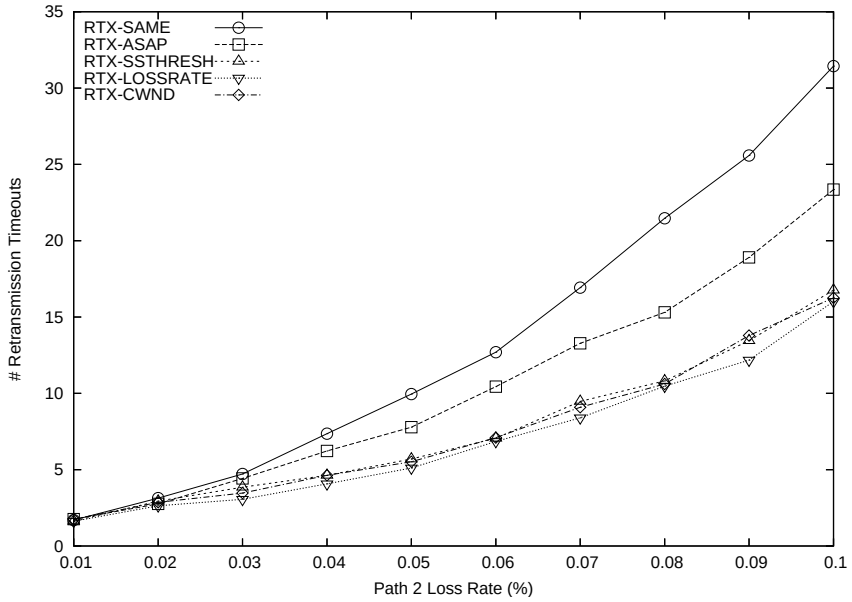
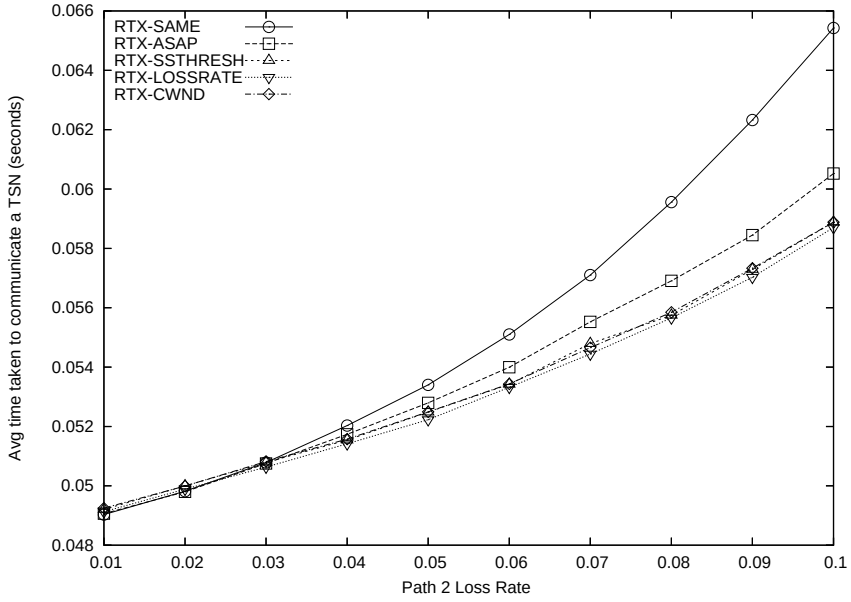
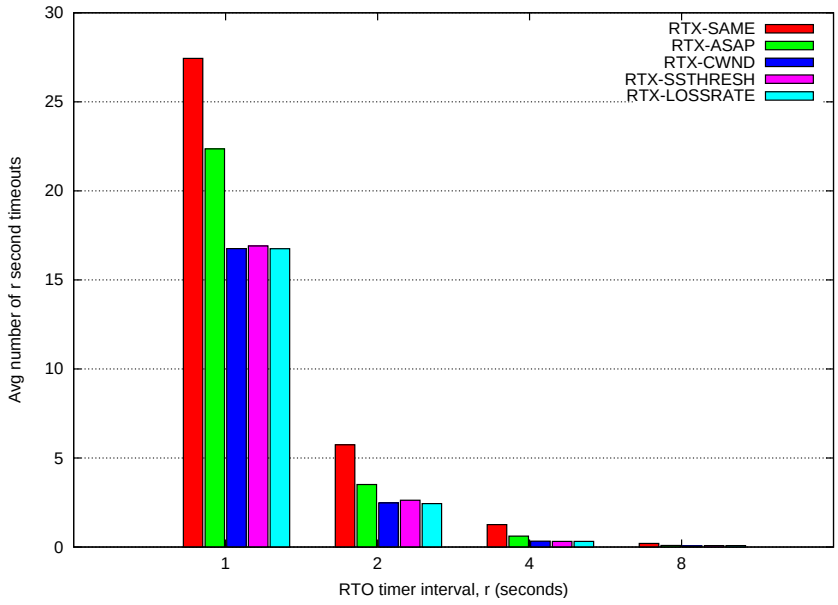


Figure 4.5: Number of retransmission timeouts for CMT with different retransmission policies (rbuf: 64K, Path 1 loss rate: 1%)

Figure 4.6(b) shows average number of timeouts that take r seconds (for r = 1, 2, 4 and 8) using the different retransmission policies with Path 1 loss rate = 1%, and Path 2 loss rate = 10%. Using SCTP's (and TCP's) default parameter values, these values of r are caused by consecutive timeouts: r = 2 seconds corresponds to 2 consecutive timeouts, r = 4 seconds corresponds to 3 consecutive timeouts, etc. Though the occurrences of 2 and 4 second timeouts are few, we note that their impact is significant due to rbuf blocking during these periods. Overall, loss-rate-based policies experience about half the consecutive timeouts that RTX-SAME does. Thus, *performance degradation due to consecutive timeouts can be significantly reduced by taking loss rate into account for making retransmission decisions.*



(a)



(b)

Figure 4.6: With rbuf = 64K, and Path 1 loss rate = 1%,: (a) Time taken to successfully communicate a TSN with different retransmission policies, (b) Consecutive timeouts with different retransmission policies (Path 2 loss rate = 10%)

4.2.2 Evaluation with Different rbufs

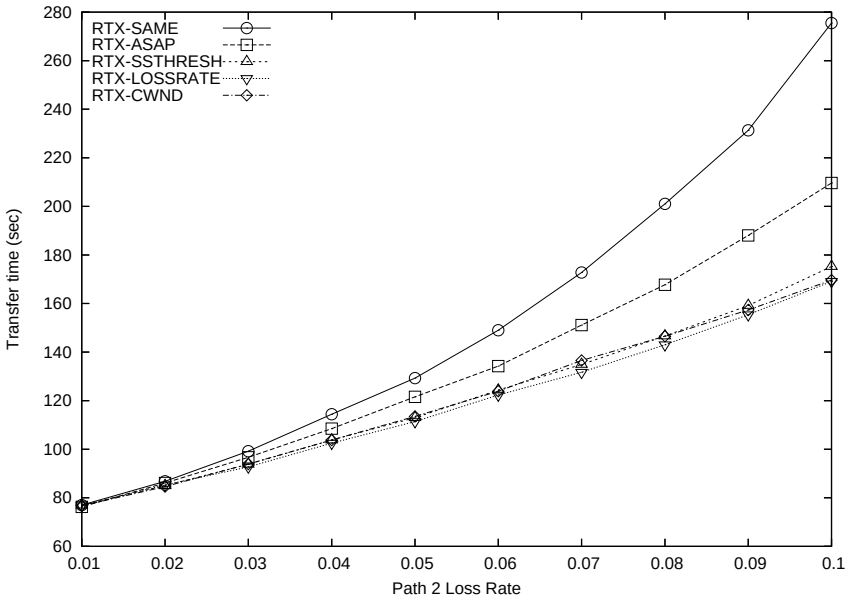
Figures 4.7 and 4.8 show the performance of the five retransmission policies using rbuf sizes of 16KB, 32KB, 128KB, and 256KB. The performance ranking of the different policies with these rbufs remains the same as with an rbuf of 64KB (Figure 4.4). We discuss a few salient points.

- With a large (i.e., minimally constraining) rbuf of 256KB, RTX-SAME still performs poorly due to a high number of timeouts, and the consequent throughput degradation. Each timeout causes cwnd reduction at a sender and entails idle time (i.e., the sender not transmitting data), causing throughput reduction.

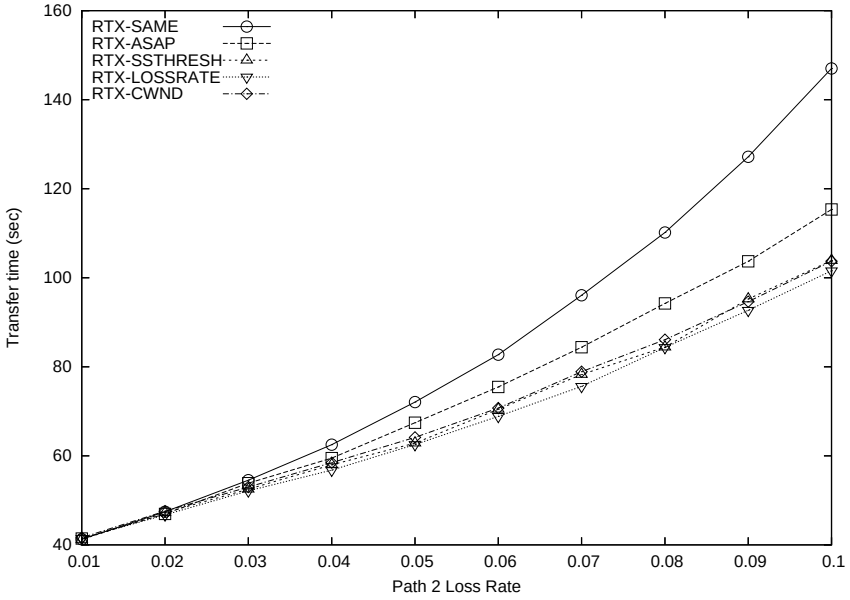
- As the rbuf size decreases and becomes more of a constraint, degradation in CMT throughput occurs due to increased rbuf blocking. All retransmission policies suffer in the face of an increasingly constraining rbuf. Even with a reasonably large rbuf of 128KB, some performance degradation occurs; i.e., *even with large rbufs, rbuf blocking is not eliminated*.

- As the rbuf size decreases and blocking increases, a decreasing fraction of data is sent on the better path since a sender can send lesser data on the better path during periods of loss recovery on the worse path. Consequently, an increasing fraction of data is sent on the worse path, causing an increasing number of losses and further degrading CMT throughput.

- As the rbuf becomes more constraining, degradation in throughput of RTX-SAME policy is markedly more than with the other retransmission policies. The reasons for this degradation are the same as described in Section 4.2.1. Degradation is least in the loss-rate-based policies even with an increasingly constraining rbuf.

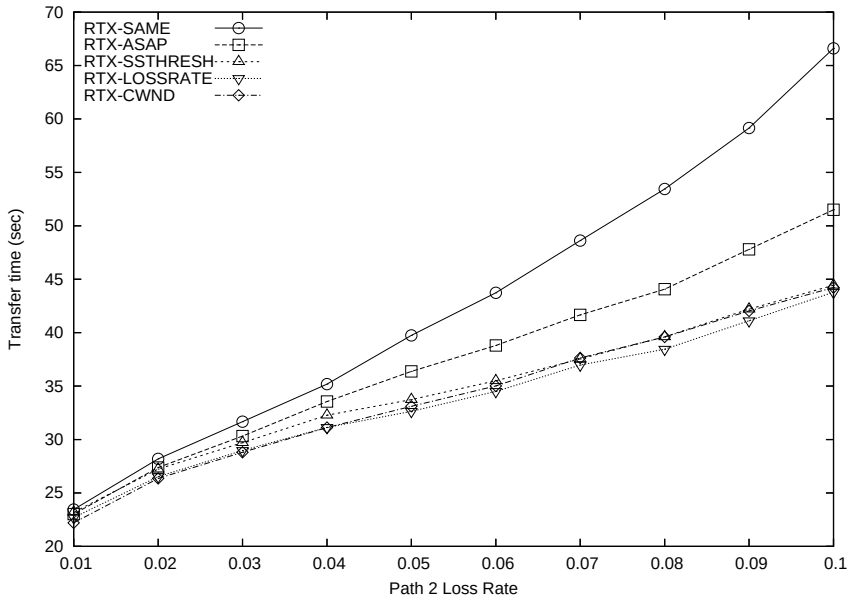


(a)

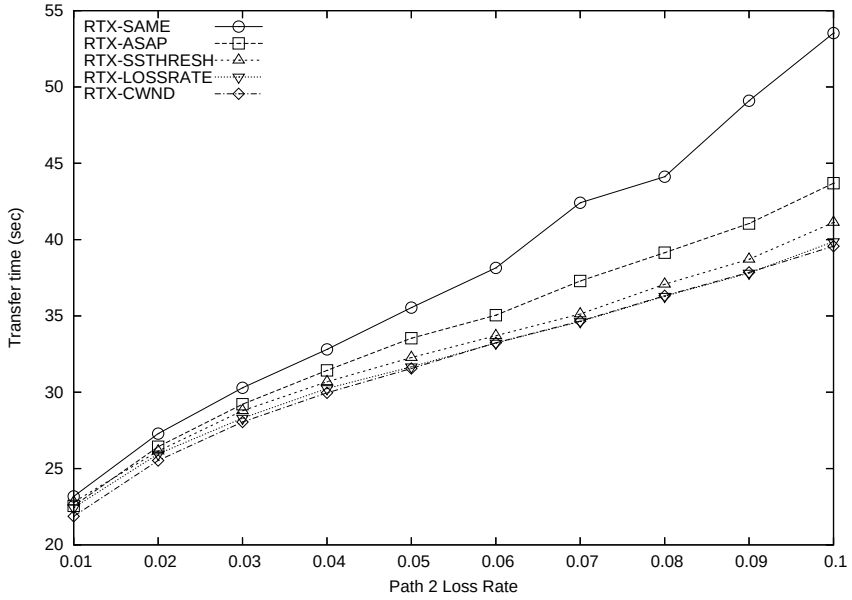


(b)

Figure 4.7: With Path 1 loss rate = 1%, time taken by CMT to transfer an 8MB file using: (a) rbuf = 16K (b) rbuf = 32K



(a)



(b)

Figure 4.8: With Path 1 loss rate = 1%, time taken by CMT to transfer an 8MB file using: (a) rbuf = 128K (b) rbuf = 256K

In summary, *retransmission policies that take loss rate into account perform better, increasingly better as rbuf size decreases.* Of the loss rate based policies, the practical ones (RTX-CWND and RTX-SSTHRESH) perform similarly under all conditions considered. *Therefore, we recommend the loss-rate-based policies, RTX-SSTHRESH and RTX-CWND, for CMT.* We arbitrarily choose RTX-SSTHRESH as CMT's retransmission policy in further evaluations.

4.3 Performance Impact of Receive Buffer Blocking

In this section, we study the impact of rbuf blocking on CMT under different network conditions. We explore the effect of different end-to-end delays (Section 4.3.1), and different combinations of delays and loss rates (Sections 4.3.2 and 4.3.3) on CMT's throughput.

4.3.1 Performance Under Different Equal End-to-end Delays

Figure 4.9 shows relative throughput degradation of CMT under different end-to-end delays - 10ms, 25ms, 45ms, 90ms, 180ms, and 360ms, yielding RTTs of 20ms, 50ms, 90ms, 180ms, 360ms, and 720ms, respectively. The delays on both paths to the receiver are symmetric (rbuf blocking with asymmetric paths is studied later in Sections 4.3.2 and 4.3.3). These values cover a range of RTTs experienced by majority of flows on the Internet [58]. Relative throughput degradation is computed as the ratio

$$\frac{CMT\ throughput\ with\ infinite\ (INF)\ rbuf}{CMT\ throughput\ with\ rbuf = X} \tag{4.1}$$

as X varies from 16KB to 256KB along the X-axis (Note that along the Y-axis, smaller values are better). The largest degradation (i.e., worst throughput) occurs with the shortest delay of 10ms. At 10ms, throughput with infinite rbuf space is roughly 10 times what it would be with a 16KB rbuf. As the end-to-end delay increases, CMT's relative throughput

degradation decreases. We now explain why associations with smaller delays are more sensitive to a constrained rbuf.

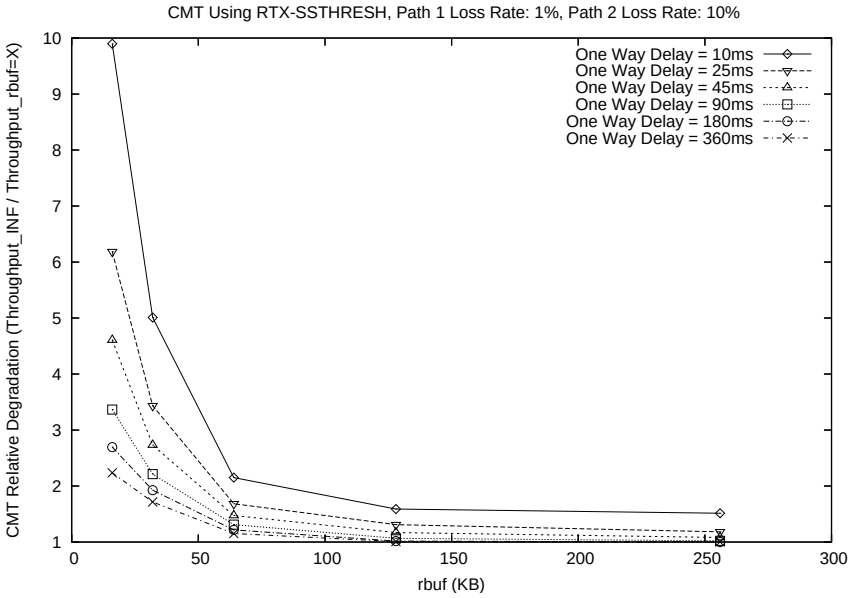


Figure 4.9: Relative throughput degradation of CMT with different end-to-end delays

Overall SCTP throughput, similar to TCP throughput, varies inversely with delay. This relationship holds true for large rbuf conditions. Thus, in the relative throughput degradation measure, the numerator (CMT throughput with infinite rbuf) increases as delay decreases.

As the rbuf size increasingly becomes a bottleneck, a different dynamic dominates. According to the SCTP specification [65] and the specification for computing TCP's retransmission timer [54], retransmission timeouts (RTOs) should have a (conservative) minimum value of 1 second to avoid spurious timeouts. These timeout recovery periods are thus independent of the end-to-end delays considered, since the delays are far less than 1 second. As rbuf increasingly constrains the traffic flow, the number of timeouts increases. Consequently, total time spent in timeout recovery (which is roughly the same

irrespective of the end-to-end delay) increasingly dominates association lifetime. Thus, with decreasing end-to-end delay, the denominator in equation (4.1) does not increase as fast as the numerator, since the denominator is largely dictated by (constant) timeout recovery periods. Therefore, the influence of a constrained rbuf increases as end-to-end delay decreases. In summary, *CMT is more sensitive to rbuf constraints in environments with shorter end-to-end delay (such as data centers [42]). Or, from a network engineering point of view, the shorter the end-to-end delay, the more important it is to have a larger rbuf to fully exploit CMT.*

We can thus see that rbuf blocking has a larger impact on associations with shorter end-to-end delay due to a minimum RTO value which is recommended [54,65] and largely in use. We note that shorter minimum RTOs together with better RTT estimation algorithms [14, 25] may lessen the bias against shorter delay associations.

4.3.2 CMT vs. UnawareApp

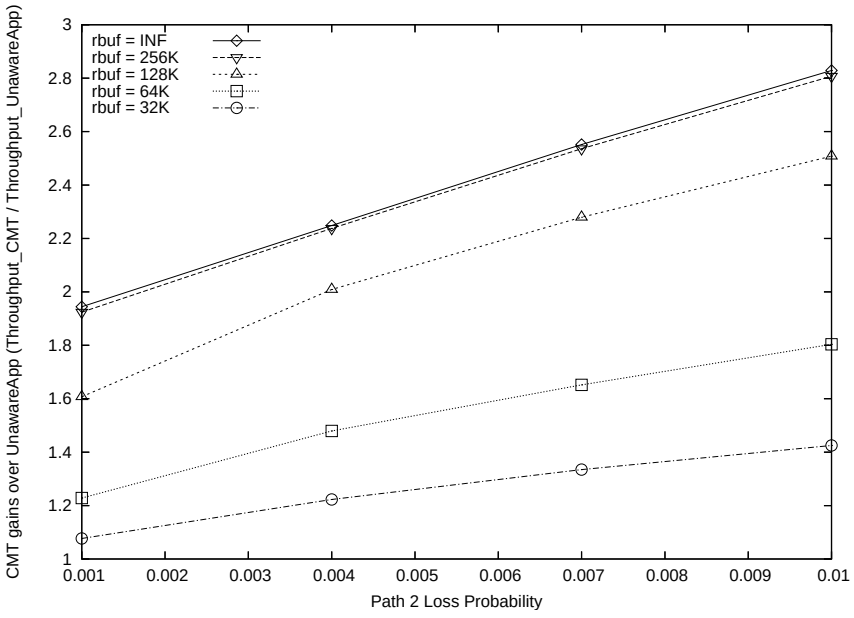
The potential parallelism gains of CMT decrease as the rbuf size decreases. In this section we attempt to quantify the gains, if any, in using CMT with a limited rbuf. We introduce a reference for comparison called *UnawareApp* and compare it to CMT. UnawareApp represents the expected throughput seen by an application using a single SCTP association to transfer data. UnawareApp sends data to one destination selected from the set of receiver destinations with equal probability.

It might seem unintuitive to compare CMT against UnawareApp, since UnawareApp uses just one path for the transfer (unlike CMT which sends data to all destinations). To address this concern, we point out that this evaluation explores the impact of the rbuf blocking problem on CMT—a goal in this evaluation is to see if rbuf blocking can degrade CMT's throughput to the extent that UnawareApp outperforms CMT. The rbuf blocking problem does not exist for data transfers that use only one path, and therefore, does not affect

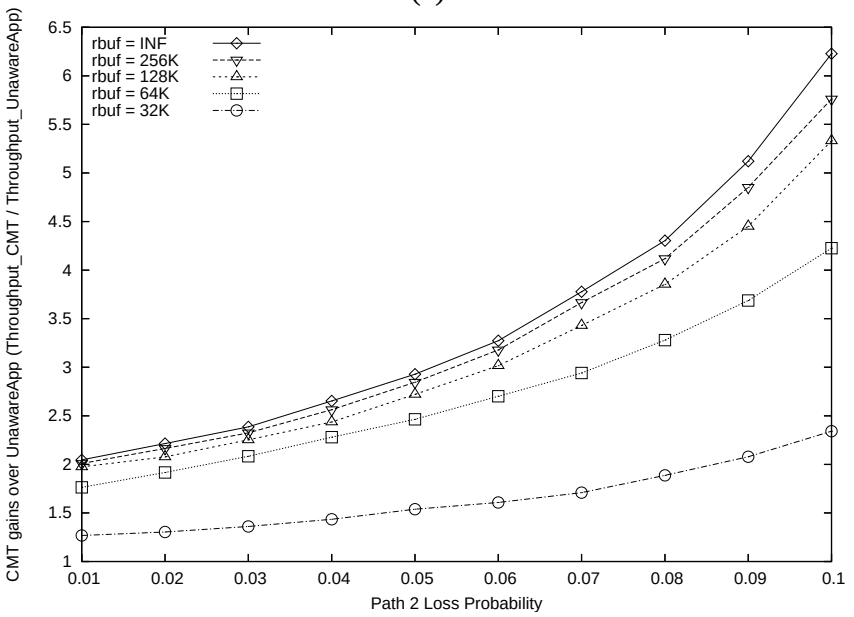
UnawareApp. Further, without prior knowledge of path conditions, and without CMT, an application would arbitrarily pick one destination to send data. UnawareApp captures the expected throughput when such a decision is made.

We do not compare CMT with application level load sharing such as AppStripe (Section 3.2) because with a constrained receive buffer, AppStripe ceases to be an *ideal case* to compare against. Ideal scheduling decisions are needed to use the finite buffer space *ideally*. We believe that devising such an ideal scheduler at the application layer is impractical. If practical, then the same scheduling decisions can be used in CMT. Further, rbuf blocking is caused by a shared finite receive buffer, and equally affects any data striping application such as AppStripe. An application that stripes data across multiple paths requires a finite rbuf at the application layer for reassembly and ordering of incoming data. This rbuf will cause analogous degradation at the application layer as a bounded transport layer rbuf causes for CMT. Therefore, we do not use a data striping application as a reference in this evaluation since the rbuf blocking problem exists for such an application as well. On the other hand, CMT will have performance gains over AppStripe due to sharing of sequence space across paths (Section 3.4) and intellgent redirection of retransmissions. Therefore, we argue that CMT will perform better than an Appstripe implementation with finite receive buffer.

Figures 4.10 and 4.11 shows throughput gains in using CMT vs. UnawareApp, measured as a ratio of CMT throughput to UnawareApp throughput, for different rbuf values, different loss rates on the two paths, and different delay combinations on the two paths. We explored loss rate combinations in the range [0.1%, 10%], end-to-end delay combinations in the range [25ms, 360ms], and rbuf sizes from 32K to 256K. These loss rate and delay combinations reflect network conditions experienced by flows on the Internet [58]. UnawareApp uses the same rbuf size as CMT in these evaluations.

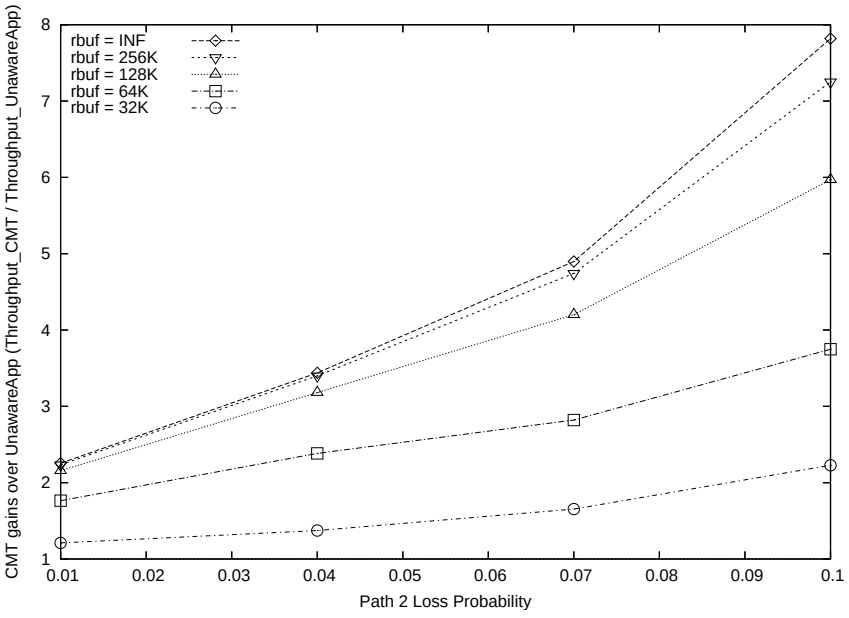


(a)

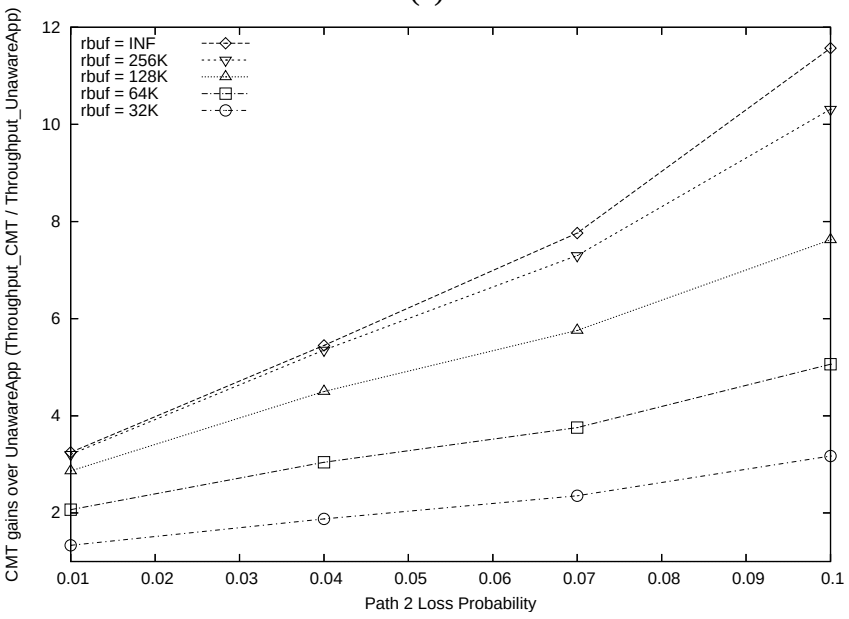


(b)

Figure 4.10: CMT throughput gains over UnawareApp with a constraining rbuf:
(a) Path 1 loss probability = 0.001, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 45ms
(b) Path 1 loss probability = 0.01, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 45ms



(a)



(b)

Figure 4.11: CMT throughput gains over UnawareApp with a constraining rbuf:
(a) Path 1 loss probability = 0.01, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 90ms
(b) Path 1 loss probability = 0.01, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 180ms

The ratio plotted in Figures 4.10 and 4.11 is

$$\frac{Throughput\_CMT}{Throughput\_UnawareApp} \tag{4.2}$$

Values greater than 1 imply that CMT performs better than UnawareApp; a value less than 1 means that UnawareApp performs better than CMT. All results show similar trends in throughput with CMT performing better; representative results are shown in Figures 4.10 and 4.11. The throughput gains with CMT are chiefly attributed to two reasons:

- Since our topology has two paths between the sender and the receiver, UnawareApp chooses the worse path for transferring data half the time. Transfer times over the worse path increase significantly as loss rate on that path increases, thereby increasing the average transfer time for UnawareApp significantly. CMT uses both paths concurrently, and using RTX-SSTHRESH ensures that most of the retransmitted data is sent over the better path, thus reducing overall transfer time for CMT.
- CMT is more resilient to reverse path loss than UnawareApp. CMT uses a single sequence space across the one association's multiple paths. Since CMT's acks are cumulative, sharing of sequence spaces across paths helps a CMT sender receive ack info on either of the return paths.

These results demonstrate that though rbuf blocking degrades CMT's throughput, using CMT is still beneficial, even with the most constrained rbuf of 32KB; this benefit increases as the rbuf size increases.

4.3.3 CMT vs. AwareApp

By using UnawareApp as a reference, we assumed that a sender has no prior knowledge of path conditions. This assumption causes UnawareApp to suffer throughput degradation,

since UnawareApp picks and uses the higher loss rate path half the time (in our simulations, two paths are used). We now drop this assumption, and introduce *AwareApp*, an application which has *a priori* path information, and uses always the lowest loss rate path for data transfer. AwareApp represents an application's throughput when using a single SCTP association over the best path to the destination. AwareApp avoids the rbuf blocking problem (as does UnawareApp), and also avoids throughput degradation due to using the higher loss rate path. A goal in this evaluation is to see if rbuf blocking can degrade CMT's throughput to the extent that using only the better path (AwareApp) outperforms using both paths (CMT).

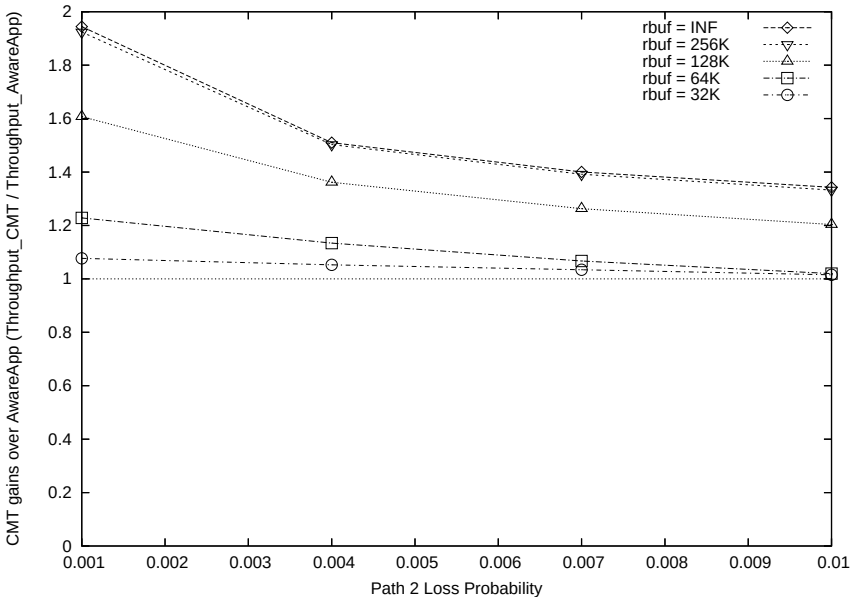
As in Section 4.3.2, we explored different combinations of loss rate in the range from 0.1% to 10%, and end-to-end delays in the range from 25ms to 360ms with rbuf values ranging from 32KB to 256KB. Representative results from our exhaustive set of simulations are shown in Figures 4.12 and 4.13. The ratio plotted in these figures is

$$\frac{Throughput\_CMT}{Throughput\_AwareApp} \tag{4.3}$$

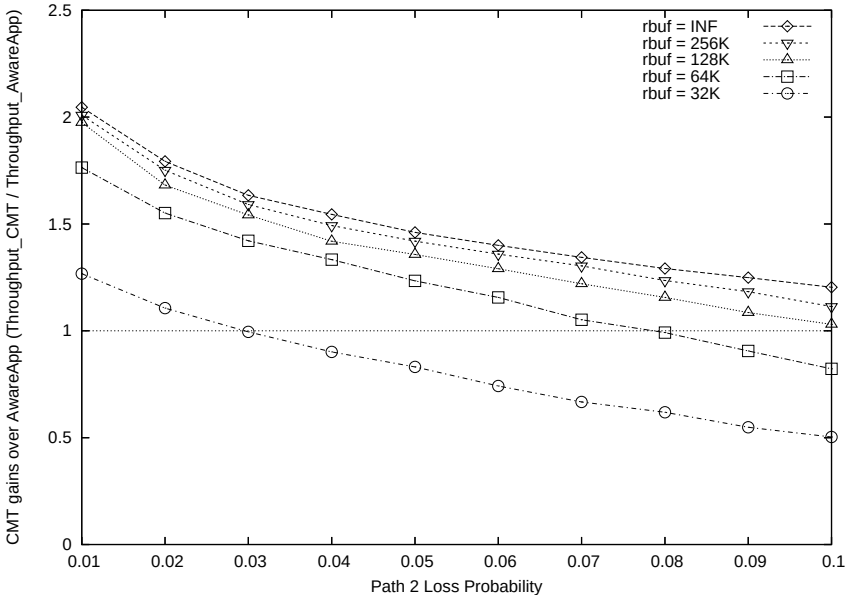
A value greater than 1 implies that CMT performs better than AwareApp; a value less than 1 means that AwareApp performs better than CMT. Salient points are as follows:

- *In some cases, AwareApp performs better than CMT*. These cases can be seen in Figures 4.12 and 4.13 whenever the curves drop below 1. This result is significant—rbuf blocking can degrade throughput to the point that *when large differences exist in path delays and loss rates, using only the better path outperforms using two paths concurrently*.

- CMT's throughput benefit over AwareApp decreases as the loss rate difference between the two paths increases for two — increased losses, and increased rbuf blocking. On the other hand, AwareApp, which uses only the lower loss rate path (i.e., Path 1), does not experience either of these throughput degradations.

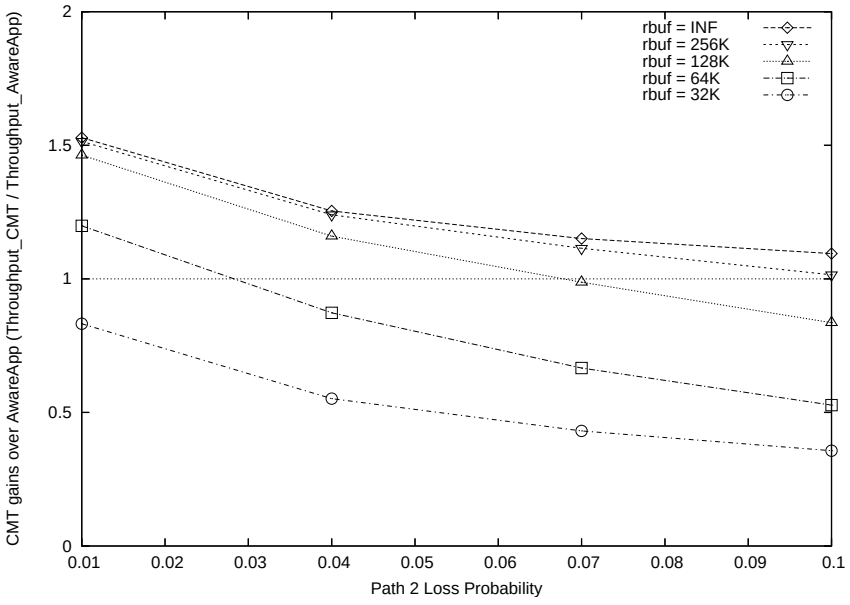


(a)

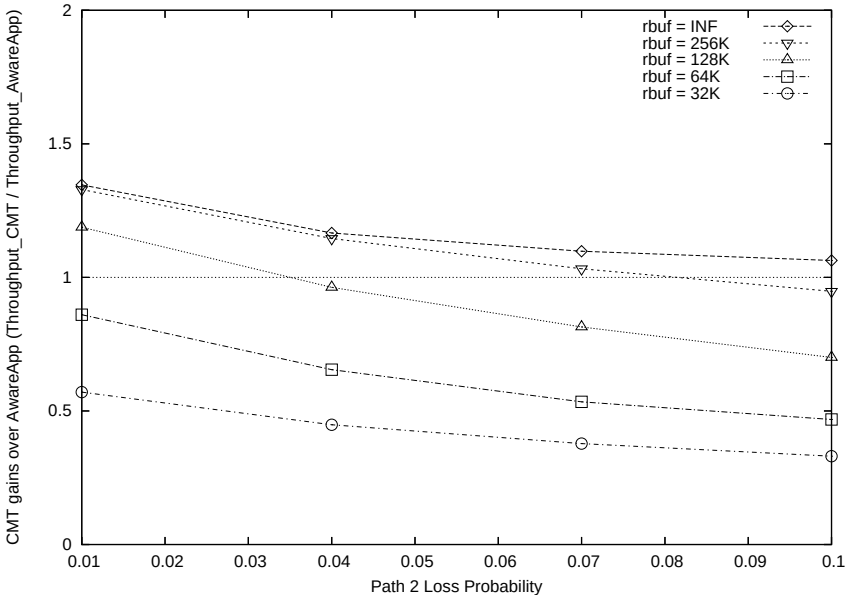


(b)

Figure 4.12: CMT throughput gains over AwareApp with a constraining rbuf:
(a) Path 1 loss probability = 0.001, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 45ms
(b) Path 1 loss probability = 0.01, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 45ms



(a)



(b)

Figure 4.13: CMT throughput gains over AwareApp with a constraining rbuf:
(a) Path 1 loss probability = 0.01, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 90ms
(b) Path 1 loss probability = 0.01, Path 1 end-to-end delay = 45ms, Path 2 end-to-end delay = 180ms

- CMT's throughput benefit over AwareApp decreases, although not significantly, as the delay difference between the two paths increases. See Figures 4.12(b), 4.13(a), and 4.13(b), where Path 2 delay is 45ms, 90ms, and 180ms, respectively (Path 1 delay is maintained at 45ms). As the delay of Path 2 increases, more data can be sent on Path 1 within one roundtrip time on Path 2 filling up the rbuf, thereby causing more rbuf blocking even when no loss occurs. An increase in Path 2's end-to-end delay also increases occurrences and periods of rbuf blocking due to increased loss recovery time on Path 2, for fast retransmit based recovery. This increased rbuf blocking degrades CMT's throughput.

We conclude that rbuf blocking degrades performance increasingly with increasing difference in path loss rate and delay. Therefore, *the larger the difference between the paths, the larger the rbuf required to avoid degradation due to rbuf blocking.*

Though the conditions studied represent Internet conditions [58], we acknowledge that we are using limited and simplistic simulations to extract an exact value for use in real complex networks; we therefore *suggest with caution* that a minimum rbuf of 128KB is required to exploit CMT's parallelism. This suggestion is only meant to provide a "ballpark" value when using CMT, and is not a strong conclusion. We strongly encourage CMT users to experiment with rbuf sizes in their deployment scenarios—more experience with CMT will provide a better understanding of what rbuf size should be used in specific deployment scenarios.

4.4 Evaluation With Cross-traffic Based Losses

A valid criticism of the results thus far is that they have been based on the simple topology in Figure 4.1. We now verify a subset of our results using a more realistic simulation

topology with cross-traffic[3]. Our goal is to observe the relative performance of the different retransmission policies with a traffic model that better resembles observed traffic on today's Internet [51].

Figure 4.14 shows the network topology: a dual-dumbbell topology whose core links have a bandwidth of 10Mbps and a one-way propagation delay of 25ms. Each router, R, is attached to five edge nodes. One of these five nodes is a dual-homed node for a SCTP endpoint, while the other four are single-homed and introduce cross-traffic.

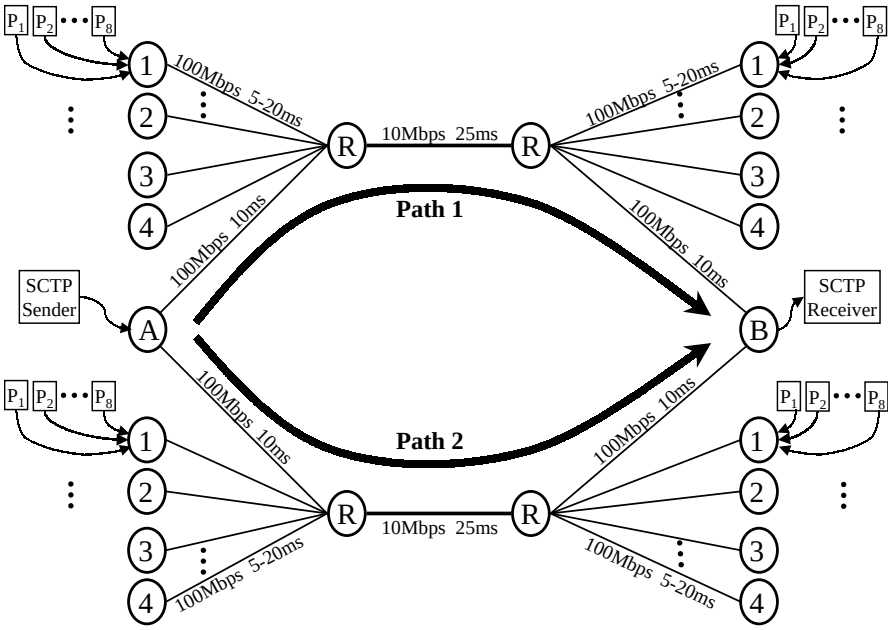


Figure 4.14: Simulation network topology with bursty cross-traffic and congestion loss

The links to the dual-homed nodes have a bandwidth of 100Mbps and a one-way propagation delay of 10ms. The single-homed nodes also have 100Mbps links, but their propagation delays are randomly chosen from a uniform distribution between 5-20ms to simulate

[3] We note that the full set of simulations performed for the results thus far presented in this chapter using the model in Figure 4.1 would have taken too long to achieve.

end-to-end one-way propagation delays between 35ms and 65ms. These delays approximate Internet delays for distances such as coast-to-coast of the continental US, and eastern US to/from western Europe. Also, each link (both edge and core) has a buffer size twice the link's bandwidth-delay product, which is a reasonable setting in practice.

Figure 4.14 has two SCTP endpoints with CMT (sender A, receiver B) on either side of the network, which are attached to the dual-homed edge nodes. A has two paths, labeled Path 1 and Path 2, to B. The observed nature of aggregate traffic on data networks is self-similar [51], and can be modeled as an aggregation of ON/OFF sources with durations drawn from distributions with heavy tails (e.g., Pareto) [70]. Therefore, each single-homed edge node has eight traffic generators, each exhibiting ON/OFF patterns with ON-periods and OFF-periods drawn from a Pareto distribution. The cross-traffic packet sizes are chosen to *roughly* resemble the distribution found on the Internet: 50% are 44 bytes, 25% are 576 bytes, and 25% are 1500 bytes [3, 22].

We simulate a 32MB file transfer with different network conditions, controlled by varying the load introduced by cross-traffic. We increase the filesize from the 8MB used in previous experiments because transfer time with crosstraffic, due to increased burstiness in loss events, shows greater variance than when using a Bernoulli loss model. Compensating for this increased variance by increasing the number of simulation runs is expensive, since some runs take on the order of 10s of minutes to finish. A larger filesize reduces the observed variance in measured file transfer time, at little additional cost. All loss experienced is due to congestion at the routers. The aggregate levels of cross-traffic on Path 1 are maintained at 4Mbps, and on Path 2, range from 4Mbps to 10Mbps. Although we independently control the levels of cross-traffic on each of the core links, the controls for the cross-traffic on each forward-return path pair are set the same. The rbuf is sized at 64KB. All results are averages over 60 runs.

Figure 4.15 shows time taken to transfer a 32MB file using the different retransmission

policies under varying loads on Path 2. Our goal in this simulation is to observe the relative performance of the different retransmission policies using different rbuf sizes under increasing cross-traffic load (and consequently, increasing loss rate).

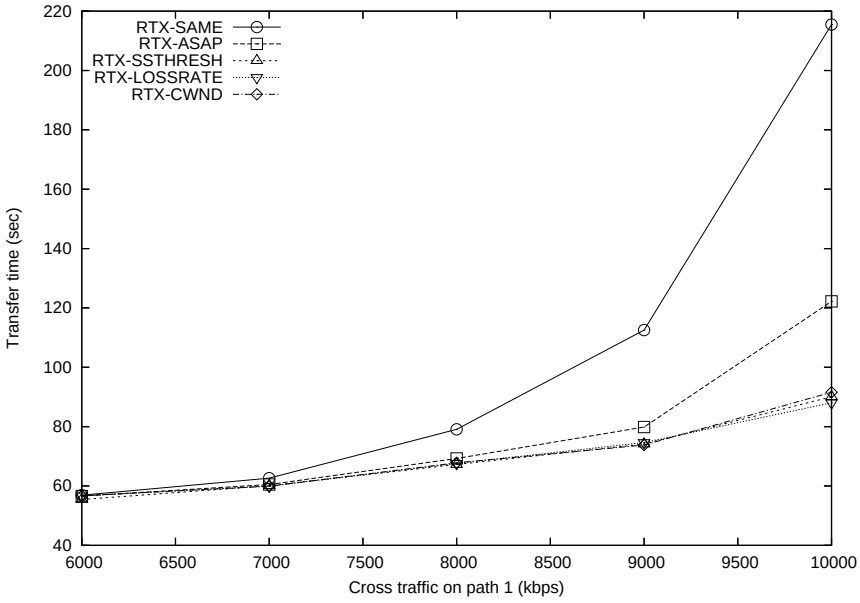


Figure 4.15: Transfer time for a 32MB file with bursty cross-traffic
(Path 1 load = 4Mbps, rbuf = 64KB)

Performances of the policies differ clearly for cross-traffic load on Path 2 greater than 7Mbps, i.e., when bottleneck link utilization is 70% or more. Generally, high levels of congestion are needed for RTOs to occur in SCTP (and TCP); therefore, performance differences (due to rbuf blocking) occur only at medium/high levels of bottleneck link utilization. The results agree with our previous conclusions about the retransmission policies—*loss-rate-based policies are equally the best performing policies*.

4.5 Discussion

As just discussed, a constrained rbuf can cause significant throughput degradation when multiple paths are used concurrently. One might expect that blocking can be avoided by multiplexing over paths at a different layer, but we note that *the rbuf blocking problem cannot be eliminated at any layer*; it can only be reduced. This problem equally affects an application layer (or network layer) data striping mechanism. Reservation of rbuf space per path will also not reduce blocking due to the need for in-order delivery to the application. As we have shown, use of an intelligent retransmission policy, which is possible in only the transport layer, and/or using a larger rbuf reduces rbuf blocking.

In addition to a receive buffer (rbuf) at the receiver, a transport sender maintains a send buffer (sbuf) for two reasons: (i) to buffer data received from a sending application that sends faster than can be transmitted into the network, and (ii) to maintain data that has been sent, but not cumulatively acked, for possible retransmission. While we discuss only rbuf throughout this chapter, we note that these evaluations, analyses and recommendations apply to both sender and receiver socket buffers. If the sbuf is blocked by a higher loss rate path in a CMT association (as the rbuf can be), i.e., no data can be sent from the sending application to the sending transport, then the CMT association will suffer from the same degradation as described in this chapter. Our suggestion of using a minimum rbuf size of 128KB, therefore, holds for the sbuf as well. Using multistreaming or unordered data delivery will allow a transport receiver to deliver to the application data that may not be globally in-order within an association. Such "out-of-order" delivery will reduce rbuf blocking, but will not affect blocking of the sbuf, since loss recovery mechanisms use global ordering within an association. In using a single ordered stream in our evaluations, we capture the effective blocking (rbuf or sbuf) that will degrade any CMT association.

With CMT, further gains can be had over UnawareApp and AwareApp (which use a single

SCTP association) in fault tolerance as well. Fault tolerance is a major motivation for, and benefit of, the multihoming feature in SCTP. In case of a network or path failure, an SCTP sender can *failover* to a different destination for sending data to a multihomed receiver. An SCTP sender normally sends data to only one receiver destination (called *primary destination*), and gathers information about paths to all other receiver destinations (called *alternate destinations*) through infrequent probes called *heartbeats*. Since these probes are infrequent, an SCTP sender may have stale or inadequate information about the alternate paths to a receiver. Throughput degradation occurs when a sender uses such information about alternate paths [19].

An SCTP sender sends new data to only a single primary destination, and is therefore unable to make informed decisions about which destination to use for data transmission in case of a network failure. A CMT sender avoids this problem because data sent concurrently on all paths act as regular and frequent probes, reflecting current conditions of all paths to a receiver. A CMT sender has more accurate information about *all* paths to a receiver, which better assists a CMT sender in detecting and responding to network failure events.

Chapter 5

CMT IMPLEMENTATION IN BSD

In this chapter, we discuss our incorporation of CMT in BSD-SCTP, which is the implementation of SCTP for the BSD family of operating systems (FreeBSD, OpenBSD, NetBSD, and Darwin). The CMT implementation work was a joint effort between this author and Randall Stewart, the primary author of the SCTP specification (RFC2960) and of BSD-SCTP.

We first outline changes that were made to BSD-SCTP to incorporate CMT (Section 5.1) and discuss problems encountered in BSD-SCTP due to ack reordering introduced by CMT (Section 5.2). We then evaluate our implementation over an emulated network (Section 5.3).

5.1 Implementation Details

BSD-SCTP (with CMT) is freely available as part of the KAME project [2]. While BSD-SCTP works with all BSD systems, our development and testing of CMT used only the FreeBSD operating system. We implemented the SFR, CUCv2 and DAC algorithms, the RTX-SSTHRESH retransmission policy, and the RTT heuristic in BSD-SCTP.

CMT can be turned ON (1) or OFF (0) using the *net.inet.sctp.cmt_on_off sysctl* switch. To allow for continued experimentation, the DAC algorithm also can be turned ON or OFF using the *net.inet.sctp.cmt_use_dac sysctl* switch. After spending about 200 hours,

we were able to incorporate CMT into BSD-SCTP with roughly 100 lines of integrated code.

- The *SFR algorithm* (Section 2.2) is implemented primarily in *sctp_indata.c:sctp_handle_segments()*, where causative TSNs are identified, and in *sctp_indata.c: sctp_strike_gap_ack_chunks()*, where additional checks are introduced when incrementing missing report counts for TSNs (as per Figure 2.3).

- The *CUCv2 algorithm* (Sections 2.3 and 3.3.1) is implemented primarily in *sctp_indata.c:sctp_handle_segments()*, where the pseudo_cumacks are tracked. Cwnd updates are handled according to the pseudo_cumacks in *sctp_indata.c:sctp_handle_sack()*.

- The *DAC algorithm* (Section 2.4) is incremental to the SFR algorithm, with the sender side algorithm implemented primarily in *sctp_indata.c:sctp_strike_gap_ack_chunks()*, and delaying of acks, as per the receiver side DAC algorithm, implemented in *sctp_indata.c:sctp_sack_check()*. We use the highest order bit in the flags field of the SACK chunk for communicating number of data PDUs being acked to the data sender. At the data receiver, this bit is set in *sctp_output.c:sctp_send_sack()* as part of the receiver side DAC algorithm. At the data sender, this bit is read from the SACK chunk in *sctp_indata.c:sctp_handle_sack()*, and used in *sctp_indata.c:sctp_handle_sack()* and *sctp_indata.c:sctp_strike_gap_ack_chunks()*.

- The *RTX-SSTHRESH* retransmission policy (Section 3.1) is implemented in *sctp_timer.c:sctp_find_alternate_net()*, a function that finds an alternate destination address for retransmissions. A boolean parameter to the function, *highest_ssthresh*, if 1, indicates that the destination to be returned is the one with the highest ssthresh, and if 0, indicates that any active alternate destination can be returned.

- The *RTT heuristic* (Section 3.3.2) is applied in *sctp_timer.c:sctp_mark_all_for_resend()*, a function that is invoked on a timeout to mark outstanding TSNs for retransmission.

- Round-robin scheduling is implemented at the data sender in *sctp_output.c:sctp_med_chunk_output()*, which transmits pending retransmissions and new data to different destinations via *sctp_output.c:sctp_fill_outqueue()*.

- CMT variables, constants, and sysctls are declared and defined in *sctp.h, sctputil.c, sctp_structs.h, sctp_usrreq.c, and sctp_var.h*.

5.2 Concerns with Stale Acks

Stale acks are acks received later than acks for later data. Figure 5.1 illustrates an example which shows a stale ack being received by a CMT data sender. In the figure, the SACK that is received at time t_2 is stale because the SACK received at an earlier time t_1 conveys more recent SACK information. Stale acks occur in CMT due to delay differences in the ack paths to a data sender. Such stale acks are not identified by a CMT data sender, but commonly occur due to CMT-introduced reordering, and can cause unexpected side-effects.

We discuss two instances of unexpected behavior with stale acks in BSD-SCTP:

- From the data sender's point of view in Figure 5.1, TSN x is reneged (time t_2). This apparent renege causes the sender to assume that TSN x needs to be retransmitted and wait for a fast retransmission to occur (i.e., wait for 3 missing reports). But the subsequent ack that acks TSN x again (time t_3) may cause unexpected behavior because an SCTP sender does not expect to receive an ack for a reneged TSN without a retransmission—the SCTP specification (RFC2960) does not specify sender behavior in such a situation. In BSD-SCTP, such a renege causes a sender to lose

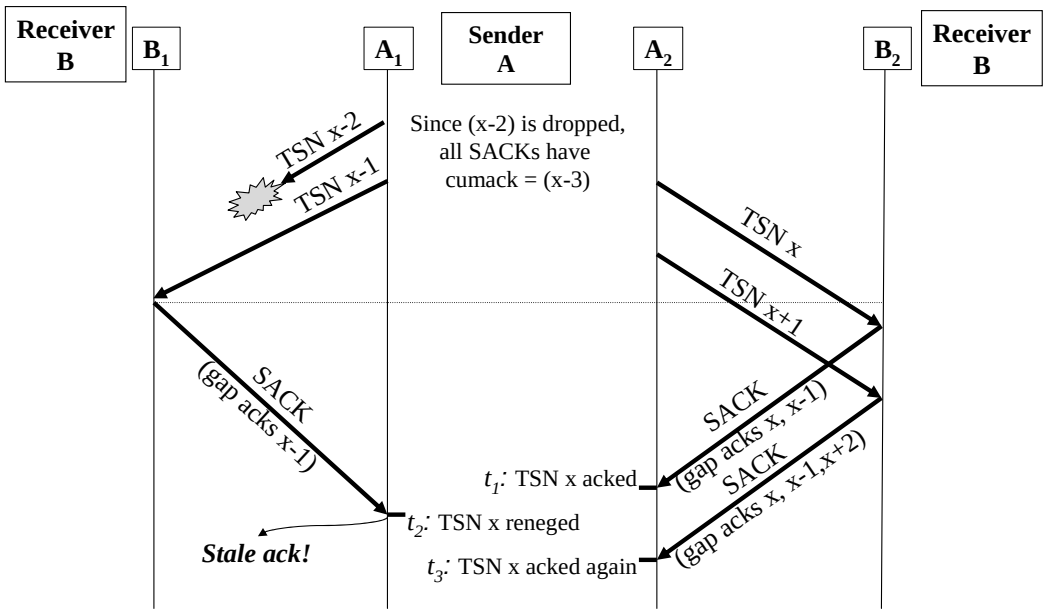


Figure 5.1: Example timeline illustrating stale acks

track of the amount of data outstanding in the network, thus causing the sender to send effectively uncontrolled amounts of data into the network. We resolved this issue for CMT by turning OFF detection of reneged TSNs through gap reports—truly reneged TSNs can be recovered through only a timeout when CMT is used. While this change will increase the recovery time for reneged TSNs from roughly one roundtrip time to one RTO, the change is conservative, and prevents spurious reneges. Further, since reneging is expected to be an infrequent occurrence, performance degradation due to this change will not be significant.

- When a TSN is newly acked, this TSN can be used to increment missing report counts on missing TSNs. When an acked TSN is incorrectly assumed reneged due to a stale ack and then subsequently acked again, the TSN may be incorrectly considered as newly acked again. Consequently, this "newly acked" TSN may allow incorrect increments of missing reports, causing early fast retransmissions.

We resolved this issue by checking if a newly acked TSN was reneged in the past before allowing it to be used for incrementing missing report counts.

We point out that it is not uncommon for the network to reorder PDUs [11], which can cause stale acks to be received with SCTP (or TCP-SACK) as well. A future work item for this author is to evaluate the side-effects of ack reordering on SCTP and TCP-SACK.

5.3 Evaluation

Using *dummynet*, we emulate different path properties in a multihomed network configuration (see Figure 5.2). Two multihomed endpoints, *reisling* and *fitou*, are connected through the dummynet router. The paths between the endpoints are maintained independent by creating separate virtual networks (VLANs) for each path using a Cisco Catalyst 2950 SX switch. In our experiments, the receiver's rbuf was set to 128KB, as suggested in Section 4.3.3 to minimize the effects of rbuf blocking. Network parameters are the same as with previous simulation experiments in Chapters 3 and 4 (described in Section 3.2). Using dummynet, end-to-end delays on the two paths were set to 45ms, representing roughly US coast-coast delays on the Internet. To observe the effects of increasing difference between path loss rates, loss rate on Path 1 was fixed at 1%, and on Path 2 was varied between 1 to 10%.

We transferred an 8MB file using (i) 1 CMT association over both paths, (ii) 1 SCTP association (denoted SCTP_1) using $fitou_1$ as primary and $fitou_2$ as alternate destination, and (iii) 1 SCTP association (denoted SCTP_2) using $fitou_2$ as primary and $fitou_1$ as alternate destination (these two SCTP transfers were sequential, and not concurrent). CMT uses the RTX-SSTHRESH retransmission policy (see Section 3.1). The retransmission policy used for SCTP is the *FrSameRtoAlt* policy (as recommended in [16, 63])—fast retransmissions are sent to the primary destination, and timeout retransmissions are sent to an

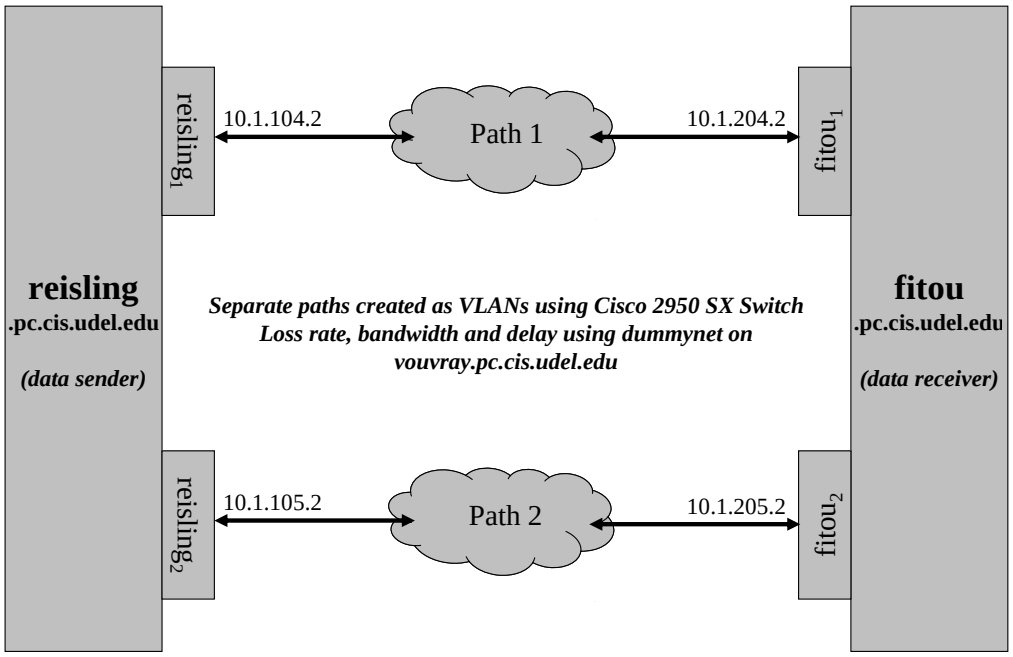


Figure 5.2: Network topology for evaluation of FreeBSD implementation

alternate destination. We note that this SCTP retransmission policy is different from the *Retransmit to Same Destination* policy used in previous SCTP simulation experiments (see Section 3.2), and is the current recommendation for SCTP [63].

Figure 5.3 shows average transfer rate when transferring an 8 MB file using CMT vs. using individual SCTP associations on each path. Figure 5.3 shows the transfer rate using the three modes of transfer discussed above. A fourth curve in the figure represents the sum of SCTP transfer rates, i.e., the calculated sum of $SCTP_1$'s and $SCTP_2$'s measured transfer rates. Each plotted point is averaged over 100 runs. CMT's transfer rate closely tracks the sum of SCTP transfer rates. We note two salient points:

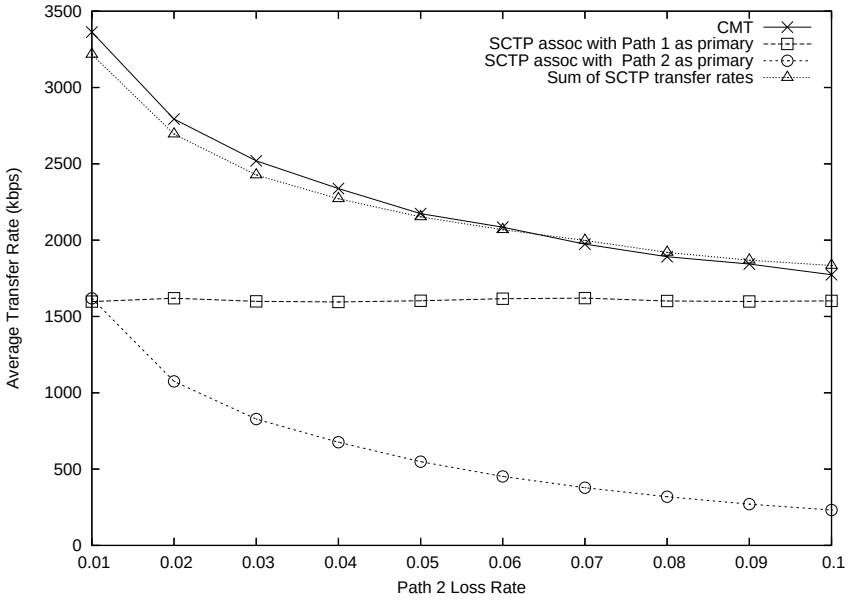


Figure 5.3: FreeBSD implementation of CMT: Performance comparison with SCTP

- CMT performs slightly better than the sum of the SCTP throughputs when the paths are symmetric, i.e., equal loss rates of 1%. Even for asymmetric paths, up to Path 2 having a loss rate of 5%, CMT does better. This improvement in CMT is due to

two reasons: (i) CMT is more resilient to ack loss than SCTP (as explained in Section 3.4), and (ii) delayed acks in slow start simultaneously contribute to the cwnd growth of both destinations in CMT. Cwnd increase in CMT occurs, therefore, at a rate higher than when delayed acks increase cwnd for the two individual SCTP associations (as explained in Section 2.4).

- As Path 2's loss rate increases, CMT's throughput gets increasingly degraded by rbuf blocking (as explained in Section 4.2).

These experiments demonstrate that our CMT implementation in BSD-SCTP performs as expected, i.e., better than the sum of individual SCTP associations when the paths are symmetric, and increasingly degraded by rbuf blocking as the asymmetry between the paths increases. This result is consistent with previous simulation results in Sections 3.4 and 4.2. We also tested CMT in BSD-SCTP with different combinations of end-to-end delays on the two paths, and again, rbuf blocking increases as asymmetry between the paths increases. The results from these experiments are consistent with simulation results in Section 4.3.3. While we specifically tested CMT using the FreeBSD operating system, we expect that other BSD and BSD-derived systems such as NetBSD, OpenBSD and Darwin will perform similarly.

Our implementation effort to incorporate CMT in BSD-SCTP was funded and encouraged by Cisco Systems, with the goal of potentially migrating CMT into their IOS operating system. We expect that incorporating CMT in BSD-SCTP will encourage wider use and experimentation with CMT in varied environments. At the time of this writing, Prof. Alan Wagner's group at the University of British Columbia, Canada, has downloaded our CMT implementation, and plan to use it for grid applications. Prof. Wagner's group is investigating an SCTP-based solution for enabling grid applications over WAN environments. In these applications, data striping is currently performed in the application (in user-space) to utilize the bandwidth of all possible network paths between hosts. CMT

enables this functionality at the transport layer (in kernel-space), and is desirable due to better performance, and reduced complexity in the application [55].

We encourage such varied use and experimentation that will contribute to better a understanding of CMT and to uncovering of hitherto unknown issues.

Chapter 6

CONGESTION WINDOW OVERGROWTH IN SCTP CHANGEOVER

In an SCTP association, a sender transmits new data to its peer's primary destination address. SCTP provides for application-initiated *changeovers* so that a sending application can redirect outgoing traffic to another path by changing the the sender's primary destination address. In this chapter, we change topics to discuss a problem discovered by this author in the current SCTP (RFC2960) specification [65] that results in unnecessary retransmissions and overgrowth of the sender's cwnd under certain changeover conditions. We present the problem here in a specific case [39]. We then develop an analytical model and discuss the results thereof [40].

This work was a precursor to this author's work on CMT, and is closely related to CMT. Since CMT can be viewed as "changeover being performed repeatedly across destinations," the changeover problem described here applies to CMT as well, as observed and described in Section 2.2. This work directly contributed to, and strongly informed our intuition, when resolving unnecessary retransmissions with CMT.

6.1 Preliminaries

The example uses a simple network topology. Endpoints A and B have an SCTP association between them. Both endpoints are multihomed, A with network interfaces A_1 and

A_2, and B with interfaces B_1 and B_2. All four addresses are bound to the SCTP association. For one of several possible reasons (e.g., path diversity, policy based routing, load balancing), independent paths are assumed. Data traffic from A to B_1 is routed through A_1, and from A to B_2 is routed through A_2. The end-to-end available bandwidth of path 1 is 10Mbps, and that of path 2 is 100Mbps. The propagation delay of both paths is 200ms, and the path MTU for both paths is 1250 bytes. The numbers chosen help to clearly illustrate the changeover problem.

Figure 6.1, shows a timeline of events for the described example. The vertical lines represent interfaces B_1, A_1, A_2 and B_2. The numbers along the lines represent times in milliseconds. Each arrow depicts the departure of a packet from one interface and its arrival at the destination. The labels on the arrows are either SCTP Transmission Sequence Numbers (TSN) or labels $ST_C(T_{GS} - T_{GE})$. Assuming one chunk per packet, every packet in the example corresponds to one TSN. A number represents the TSN of the chunk in the packet being transmitted. A label $ST_C(T_{GS} - T_{GE})$ represents a packet carrying a SACK chunk with cumulative ack T_C, and gap ack for TSNs T_{GS} through T_{GE}. C_1 is the *cwnd* at A for destination B_1, and C_2 is the *cwnd* at A for destination B_2. C_1 and C_2 are denoted in terms of MTUs, not bytes.

6.2 Congestion Window Overgrowth Problem

The sender (host A) initially sends to the receiver (host B) using primary destination address B_1. This setting causes packets to leave through A_1. Assume these packets leave the transport/network layers, and get buffered at A's link layer A_1, whereupon they get transmitted according to the channel's availability. This initial condition is depicted in Figure 6.1 at time $t = 0$, when in this example A has 50 packets buffered on interface A_1.

At $t = 1$, as TSNs 1 - 50 are being transmitted through A_1, the sender's application changes the primary destination to B_2, thus, subsequent new data from the sender is sent

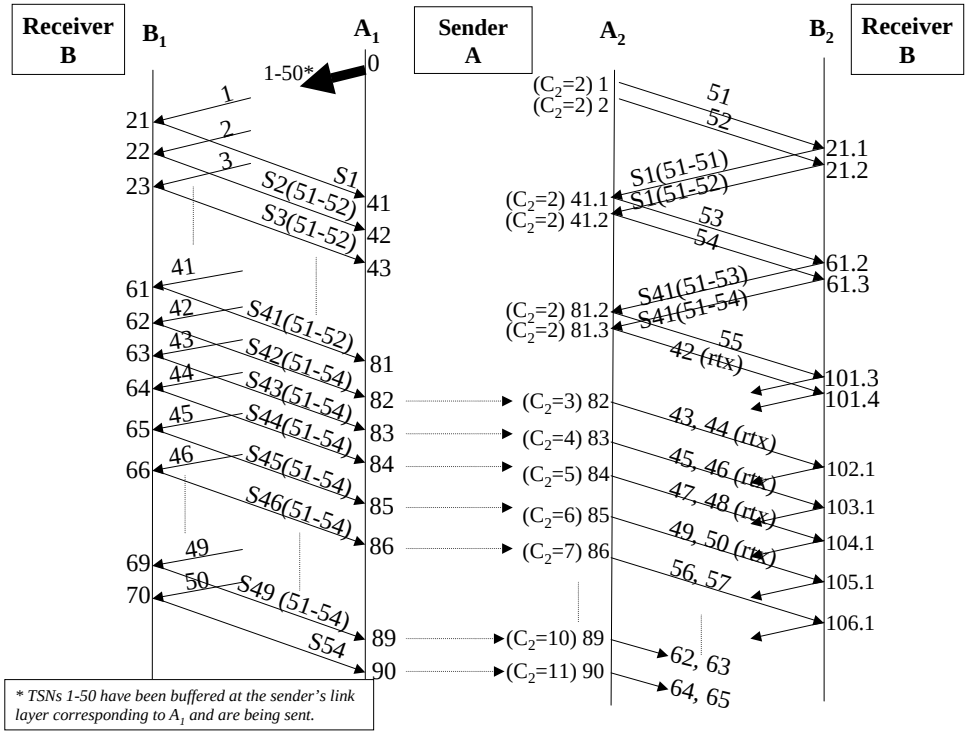


Figure 6.1: Congestion Window Overgrowth: Timeline for the example

to B_2. In the example, we assume C_2 = 2 at the moment of changeover, and TSN 51 is transmitted to the new primary at $t = 1$. We refer to this event as the *changeover time*. This new primary destination causes new TSNs to leave the sender through A_2. Concurrently, the packets buffered earlier at A_1 are still being transmitted. Previous packets sent through A_1, and the packets sent through A_2, can arrive at the receiver B in an interleaved fashion on interfaces B_1 and B_2, respectively. In Figure 6.1, TSNs 1, 51, 52 and 2 arrive at times 21, 21.1, 21.2, 22, respectively. This reordering is introduced as a result of changeover.

The receiver starts reporting gaps as soon as it notices reordering. If the receiver communicates four missing reports to the sender before all of the original transmissions (TSNs 1 - 50) have been acked, the sender will start retransmitting the unacked TSNs. SCTP's Fast Retransmit algorithm [65] is based on TCP's Fast Retransmit algorithm [8], with the additional use of selective acks and a modification to handle some cases of reordering[1]. Accordingly, the SACKs resulting from the receipt of TSNs 51-54 will be the only ones generating missing reports. The SACKs received by A on A_2 at $t = 41.1$ and $t = 41.2$ will be considered as the first and second missing reports for TSNs 2 - 50. Since these SACKs do not carry new cumulative acks, they do not cause growth in C_2. Between $t = 42$ and $t = 81$, the cumulative ack in the SACKs received by A on A_1 increases as a consequence of the original transmissions to destination B_1 reaching B. In this period, A receives 40 SACKs which incrementally carry cumulative acks of 2 - 41.

The SACKs received by A on A_2 at $t = 81.2$ and $t = 81.3$ carry a cumulative ack of 41, and are considered as the third and the fourth missing reports for TSNs 42 - 50. On the fourth missing report, A retransmits only TSN 42, since C_2 permits only one more packet

[1] [63] is an Internet Draft which goes hand-in-hand with RFC2960. The Implementor's guide maintains all changes and additions to be included in RFC2960's next version. All implementations are expected to carry the specifications and modifications in this guide.

to be outstanding. At $t = 82$, the SACK for the original transmission of TSN 42 reaches A on A_1. Since the sender cannot distinguish between SACKs generated by transmissions from SACKs generated by retransmissions, this SACK (arriving at $t = 82$) incorrectly acks the retransmission of TSN 42, thereby increasing C_2 by one, reducing the amount of data outstanding on destination B_2, and triggering the retransmission of TSNs 43 and 44. At $t = 83$, the SACK for the original transmission of TSN 43 arrives at A on A_1. As before, this SACK acks the retransmission (of TSN 43), further incorrectly increasing C_2, and triggering retransmission of TSNs 45 and 46. This behaviour of SACKs for original transmissions incorrectly acking retransmissions continues until the SACKs of all the original transmissions to B_1 (up to TSN 50) are received by A. Thus, the SACKs from the original transmissions cause C_2 to grow (possibly drastically) from wrong interpretation of the feedback.

Discussion

The values chosen in our example illustrate but a single case of the cwnd overgrowth problem. Our investigation shows that the problem occurs for a range of {propagation delay, bandwidth, MTU} settings. For example, with both paths having RTTs of 200ms (bandwidth = 100Kbps, propagation delay = 40ms) and MTU = 1500 bytes, the incorrect retransmission starts much earlier (at TSN 3), and the *cwnd* overgrowth is even more dramatic.

The cwnd overgrowth problem exists even if the buffering occurs not at the sender's link layer, but in a router along the path. In essence, the transport layers at the endpoints can be thought of as the sending and receiving entities, and the buffering could potentially be distributed anywhere along the end-to-end path.

6.3 General Model

We develop a general model for the cwnd overgrowth problem based on the example in Section 6.2. The goal of this model is to provide insight into the ambient conditions under which cwnd overgrowth can be observed, thus helping us understand the extent to which this problem can occur. This section presents a generalized timeline of SCTP behaviour during changeover, and the following sections present a derivation and discussion of analytic results leading from this general model.

We use the same topology as used in Section 6.2. The general timeline for the problem is shown in Figure 6.2. Some parameters used in the model are described below. The rest of the notation is described when referenced in Section 6.4.

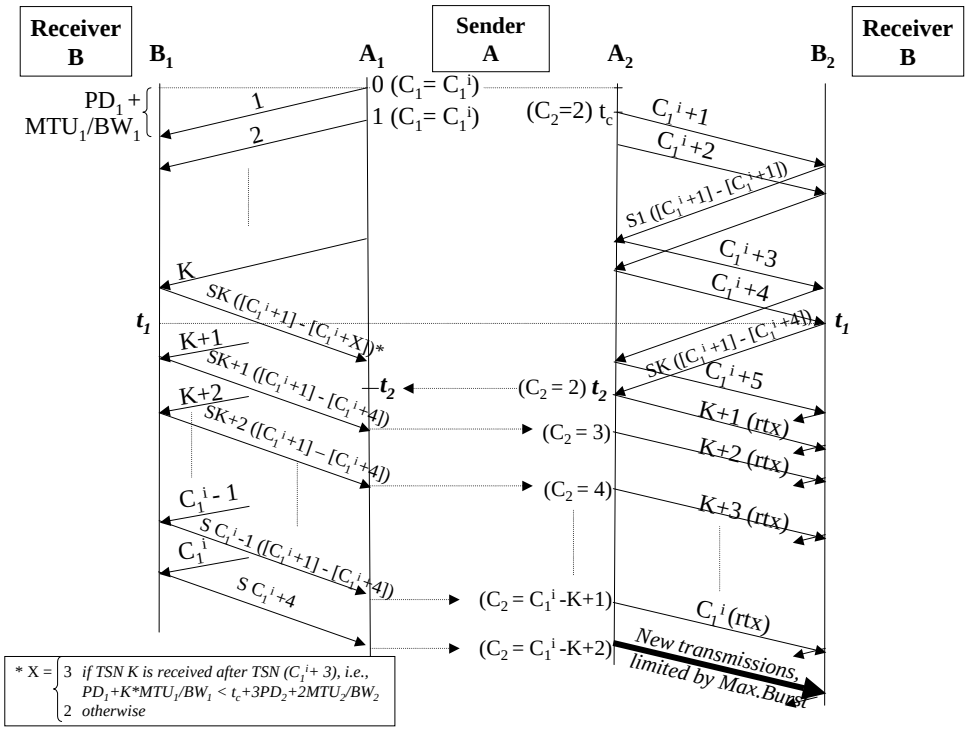


Figure 6.2: Congestion Window Overgrowth: General timeline for the problem

- C_1, C_2 : Congestion windows at A for B_1 and B_2, respectively

- t_c : Changeover time - Moment after a changeover when sender A starts sending packets to new primary destination B_2
- t_2 : Time when fast retransmission (incorrectly) starts.
- G_1 : Number of Transmission Sequence Numbers (TSNs) sent in initial group transmitted to destination B_1 in the time interval $\{0,\ t_c\}$.
- $K + 1$: First TSN to be fast retransmitted (incorrectly) by A.

At $t = 0$, host A starts to transmit G_1 TSNs (TSN 1 through G_1) to destination address B_1. By time t_c the transport layer at host A has G_1 TSNs outstanding. This group of TSNs (1 through G_1) is referred to as the *initial group*. Note that these TSNs are outstanding at the transport entity at host A, and could be buffered anywhere along the end-to-end path, even at interface A_1. By time t_c, A has changed its primary destination to B_2. At the instant $t = t_c$, A starts transmitting new data to B_2 through interface A_2. t_c can also be thought of as the time elapsed from the transmission of the first outstanding TSN on destination B_1 to the time of transmission of the first TSN on destination B_2 after changeover. Note that the SCTP receiver normally responds with delayed SACKs, but immediately returns a SACK whenever reordering is observed.

The critical instant in the scenario, denoted t_2, occurs when A receives the fourth missing report [63, 65]. At this instant, TSNs $K + 1$ through G_1 get marked for retransmission. Due to the receipt of a SACK acking TSN G_1+4, (at t_2) C_2 allows one MTU sized chunk to be transmitted, hence TSN $K + 1$ gets retransmitted to destination B_2. According to RFC2960, "... when its peer is multi-homed, an endpoint SHOULD try to retransmit a chunk to an active destination transport address that is different from the last destination address to which the DATA chunk was sent." Since the original transmission of TSN $K + 1$ went to B_1, the retransmission of TSN $K + 1$ is sent to B_2. The value of K is estimated and its relevance to the cwnd overgrowth is explained in Section 6.4.

The retransmission of TSN $K+1$ at $t = t_2$ is a consequence of the fourth missing report (SACK received on interface A_2 at $t = t_2$) carrying cumulative ack K. Since TSNs $G_1 + 1$ through $G_1 + 4$ reached host B by time t_1, the SACK also carries a gap ack for TSNs $G_1 + 1$ through $G_1 + 4$, resulting in the marking of TSNs $K + 1$ through G_1 for retransmission. The cumulative ack K is an indication that the receiver B has received K TSNs *in-sequence* by time t_1. This in-sequence data is clearly the data received by B on the interface B_1 by time t_1.

Following the retransmission of TSN $K + 1$, the SACK for the original transmission of TSN $K + 1$ arrives at A. Since host A now considers TSN $K + 1$ to be outstanding on destination B_2, the receipt of this SACK incorrectly increases C_2, and allows TSNs $K+2$ and $K+3$ to be retransmitted. The receipt of a SACK for TSN $K+1$ immediately after TSN $K + 1$ is retransmitted is not a coincidence. At time t_1 when host B sends a SACK with a cumulative ack of K acking the receipt of TSN $G_1 + 4$, TSN $K + 1$ is concurrently being received on interface B_1. Immediately after the receipt of TSN $K+1$ on interface B_1, host B sends a SACK with cumulative ack $K + 1$. Consequently, the sequence of events at host A is the receipt of a SACK with cumulative ack K (which is also the fourth missing report for TSNs $K + 1$ through G_1) followed by a SACK with cumulative ack $K + 1$. As shown, this behaviour continues until the SACKs for all the original transmissions to B_1 (up to TSN G_1) have been received at host A.

6.4 Estimation of Congestion Window Overgrowth

We will now estimate the cwnd overgrowth of C_2, and the number of unnecessary retransmissions. The parameters used in the following analysis are:

L_{1F}, L_{2F} : Maximum Transmission Unit (MTU) sizes on forward paths A_1 to B_1 and A_2 to B_2, respectively

B_{1F}, B_{2F} : End-to-End available bandwidths on forward paths A_1 to B_1 and A_2 to B_2,

respectively

e : Delay experienced by a packet along a path, given by:

$$e = \sum_{i\ =\ each\ hop} (prop)_i\ +\ (proc)_i\ +\ (queue)_i\ +\ (trans)_i \tag{6.1}$$

where $prop$ = propagation delay, $proc$ = processing delay, $queue$ = queueing delay, and $trans$ = transmission delay.

e_F : Delay experienced by a data packet, along the forward path. *Assumption:* Each data packet is MTU sized, therefore, e_F is estimated by:

$$e_F\ = \sum_{i\ =\ each\ hop\ in\ forward\ path} (prop)_i + (proc)_i + (queue)_i + \frac{L}{B^i} \tag{6.2}$$

where, L is the MTU of the path, and B^i is available bandwidth at hop i.

e_{1F}, e_{2F} : Delays experienced by a data packet on forward paths A_1 to B_1 and A_2 to B_2, respectively,

e_R : Delay experienced by a pure SACK packet, along the reverse path. *Assumption:* that transmission delays for pure SACK packets are negligible, therefore, e_R is estimated by:

$$e_R = \sum_{i\ =\ each\ hop\ in\ reverse\ path} (prop)_i\ +\ (proc)_i\ +\ (queue)_i \tag{6.3}$$

e_{1R}, e_{2R} : Delays experienced by a pure SACK packet on reverse paths B_1 to A_1 and B_2 to A_2, respectively.

d : Minimum delay observed between consecutive packets transmitted along a same path by the receiver of the packets. This delay is dictated by end-to-end available bandwidth of the path, which is determined by the hop with the minimum available bandwidth on the path (in other words, the path bottleneck). d is given by:

$$d = \frac{L}{\min_{i\ =\ each\ hop}\{B^i\}} \tag{6.4}$$

where, L is the MTU of the path, and B^i is available bandwidth at hop i.

d_{1F}, d_{2F} : Minimum delays between consecutive data packets from A_1 to B_1 observed at B_1, and from A_2 to B_2 observed at B_2, respectively.
d_{1R}, d_{2R} : Minimum delays between consecutive SACK packets from B_1 to A_1 observed at A_1, and from B_2 to A_2 observed at A_2, respectively.

Assumption: The reverse path does not change the delay between SACKs. In other words, the forward path's bottleneck dictates the rate at which SACKs are transmitted and then received, not the reverse path's bottleneck. Therefore, the delay observed *between* SACKs is the same as the delay observed between the data packets. In other words,

$$d_{1R} = d_{1F}, \ and \ d_{2R} = d_{2F} \tag{6.5}$$

Packet transmission on path 2 starts at time t_c; it takes some time for the fourth legitimate missing report to reach the sender A. This time instant is shown in figure 6.2 as t_2, which is given by:

$$t_2 = t_c + 2e_{2F} + 2e_{2R} + d_{2F} \tag{6.6}$$

t_1 is the instant when this fourth legitimate missing report *leaves* the receiver B through B_2, and is given by:

$$t_1 = t_2 - e_{2R} = t_c + 2e_{2F} + e_{2R} + d_{2F} \tag{6.7}$$

As shown in figure 6.2, we assume that the SACK received at t_2 on A_2 contains the highest cumulative ack received by A so far[2].

[2] This assumption is made for simplicity of analysis. If this assumption does not hold, the *cwnd* overgrowth will be lesser by $\lceil \frac{e_{2R}-e_{1R}}{d_{1F}} \rceil$.

Let K be defined as the TSN that was most recently cumulatively acked at A prior to time t_2. In other words, K is the last TSN that reached the receiver B on B_1 at t_1, where,

$$K = \lceil \frac{t_1 - e_{1F}}{d_{1F}} \rceil = \lceil \frac{t_c + 2e_{2F} + e_{2R} + d_{2F} - e_{1F}}{d_{1F}} \rceil \quad (6.8)$$

The result is that TSNs $(K+1)$ through G_1 will be retransmitted on Path 2 and the total number of unnecessary retransmissions = $\max\{0, G_1 - (K+1) + 1\} = \max\{0, G_1 - K\}$. The *cwnd* overgrowth for C_2 will be $\max\{0, G_1 - K\}$.

From equation (6.8), K decreases with an increase in d_{1F}, or a decrease in d_{2F}. Further, K decreases with an increase in e_{1F}, or a decrease in e_{2F}. The relationships between K and the characteristics of the two paths imply that *when a changeover is made to a higher quality path, there is a likelihood of TCP-unfriendly cwnd growth and unnecessary retransmissions, and the bigger the improvement in quality that the new path provides, the larger the TCP-unfriendly growth and number of incorrect retransmissions will be* [40].

6.5 Analytical Results: Validation and Visualization

As seen from equation (6.8), cwnd overgrowth occurs if the sender has more than K packets outstanding at the time of changeover. The value of K is thus pivotal in quantifying cwnd overgrowth. In this Section, we first validate this analytical value of K using ns-2 simulations and then estimate the value of K, using the model, under various network and changeover conditions.

6.5.1 Analytical Results: Validation

We now validate the analytical value of K derived in Section 6.4 through simulations using the ns-2 simulator. The goal in these simulations is to validate whether occurrence and number of unnecessary retransmissions match model predictions. We discuss the extent of the cwnd overgrowth problem further in Section 6.5.2.

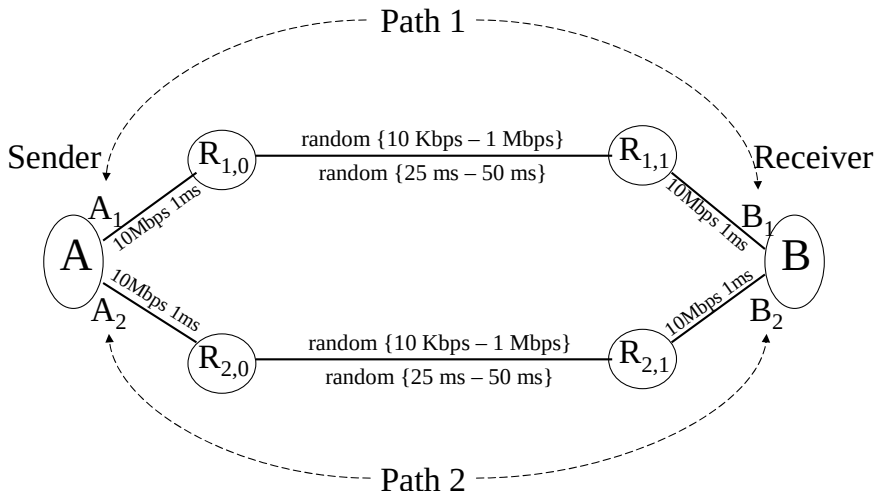


Figure 6.3: Simulation topology for validation of changeover model

We use the topology in Figure 6.3. Each of paths 1 and 2 has three links—two edge links and one core link. The edge links have a capacity of 10Mbps and propagation delay of 1ms. The available bandwidths of the paths, i.e., the capacities of the core links are chosen randomly between 10Kbps and 1Mbps. The propagation delays of the core links are chosen randomly between 25ms and 50ms. These parameters are meant to roughly represent delays that are experienced on the Internet by hosts connected via low-speed modems and/or high-speed broadband devices. The sender's sending window is fixed at 20KB by setting the receiver's advertised window to 20KB. We fix the sending window to avoid loss due to buffer overflow at the routers, and to enable easier extraction of parameters from the traces. Changeover occurs at time 5 seconds.

Of 1000 simulation runs, 511 runs showed the occurrence of incorrect fast retransmissions due to changeover. Only the runs which showed these retransmissions could be used for validation because to infer the value of K from a simulation run (denoted K_{sim}), at least one such retransmission had to occur. The first incorrect retransmission would correspond to TSN $K_{sim} + 1$.

We extracted the values of the parameters e_{1F}, e_{1R}, e_{2F}, e_{2R}, d_{1F}, d_{2F} and t_c from the traces for each of the 511 runs. Feeding these parameters into equation (6.8) gave us the analytic value of K (denoted K_{anal}).

Simulation results show that of the 511 comparisons of K_{sim} and K_{anal}, 431 results agreed exactly. In the remaining 80 results that did not agree, K_{anal} was equal to $K_{sim} - 1$. This underestimation of K by the analytic model could be attributed to the assumption made in the derivation of analytic expression for K in Section 6.4, or to approximations made in extracting the parameters from the traces.

6.5.2 Analytical Results: Visualization

In graphing the analytically derived value of K, we reduce the number of independent variables by making the following assumptions so as to visualize the graphs better:

- Forward paths 1 and 2 have the same MTU. Hence, $L_{1F} = L_{2F} = L$
- The forward and reverse paths have the same propagation, processing and queueing delays. Using equations (6.2) and (6.3),

$$e_F = e_R + \sum_{i\ =\ each\ hop\ in\ forward\ path} \frac{L}{B^i} \tag{6.9}$$

- The transmission delays at the other links along a path are assumed negligible in comparison to the transmission delay at the bottleneck link. Using equation (6.4),

$$\sum_{for\ i\ =\ each\ hop} \frac{L}{B^i} \approx \frac{L}{\min_{i\ =\ each\ hop}\{B^i\}} = d \tag{6.10}$$

- Combining the above two assumptions, we get

$$e_F = e_R + d_F = e_R + \frac{L}{B_F} \tag{6.11}$$

For the forward paths 1 and 2, the equation 6.11 can be rewritten as

$$e_{1F} = e_{1R} + \frac{L}{B_{1F}} \; and \; e_{2F} = e_{2R} + \frac{L}{B_{2F}} \tag{6.12}$$

Figures 6.5.2 and 6.5.2 (left) graph K as a function of B_{2F}, for fixed values of B_{1F}, e_{1R} and e_{2R}. In these 2-D graphs, the changeover time, t_c, is fixed at 10ms. Each 3-D graph in Figures 6.5.2 and 6.5.2 (right) picks one representative curve from the corresponding 2-D graph (left), and shows the influence of t_c on K. These 3-D graphs thus show K as a function of B_{2F} and t_c, for fixed values of B_{1F}, e_{1R} and e_{2R}.

The graphs are organized as follows:

- The results in figure 6.5.2 use the range 10kbps - 100kbps for the available bottleneck bandwidths B_{1F} and B_{2F}. t_c is set to 10ms in the 2-D graphs. The curve corresponding to B_{1F} = 50kbps is used as a representative curve to show the influence of t_c on K. t_c varies over 10ms - 100ms in the 3-D graphs. Three combinations of (e_{1R}, e_{2R}) are used: (50ms, 50ms), (50ms, 25ms), and (25ms, 50ms).

- The results in figure 6.5.2 use the range 100kbps - 1Mbps for the available bottleneck bandwidths B_{1F} and B_{2F}. t_c is set to 10ms in the 2-D graphs. The curve corresponding to B_{1F} = 500kbps is used as a representative curve to show the influence of t_c on K. t_c varies over 10ms - 100ms in the 3-D graphs. Three combinations of (e_{1R}, e_{2R}) are used: (50ms, 50ms), (50ms, 25ms), and (25ms, 50ms).

We split the range (10kbps - 1Mbps) into two subranges (10kbps - 100kbps and 100kbps - 1Mbps), because the variation observed in K with both B_{1F} and B_{2F} ranging from 10kbps to 1Mbps is large. We are thus able to visualize the behavior of K over a large range of available bandwidths, with the assumption that the available bandwidths of the two paths are comparable. In these figures, if the sender has more than K packets outstanding at

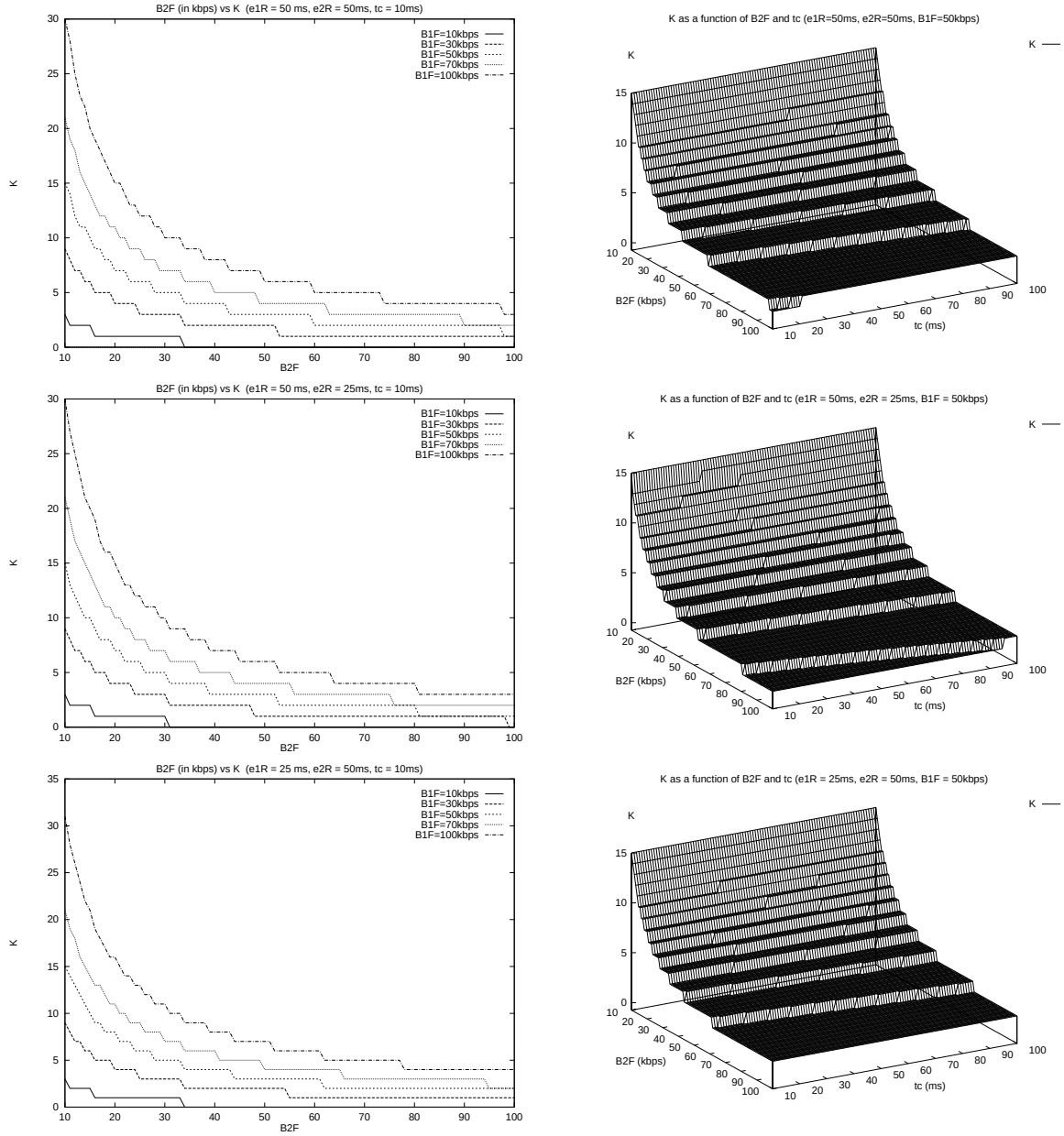


Figure 6.4: Graphing K analytically: 10kbps $\leq B_{1F}, B_{2F} \leq$ 100kbps

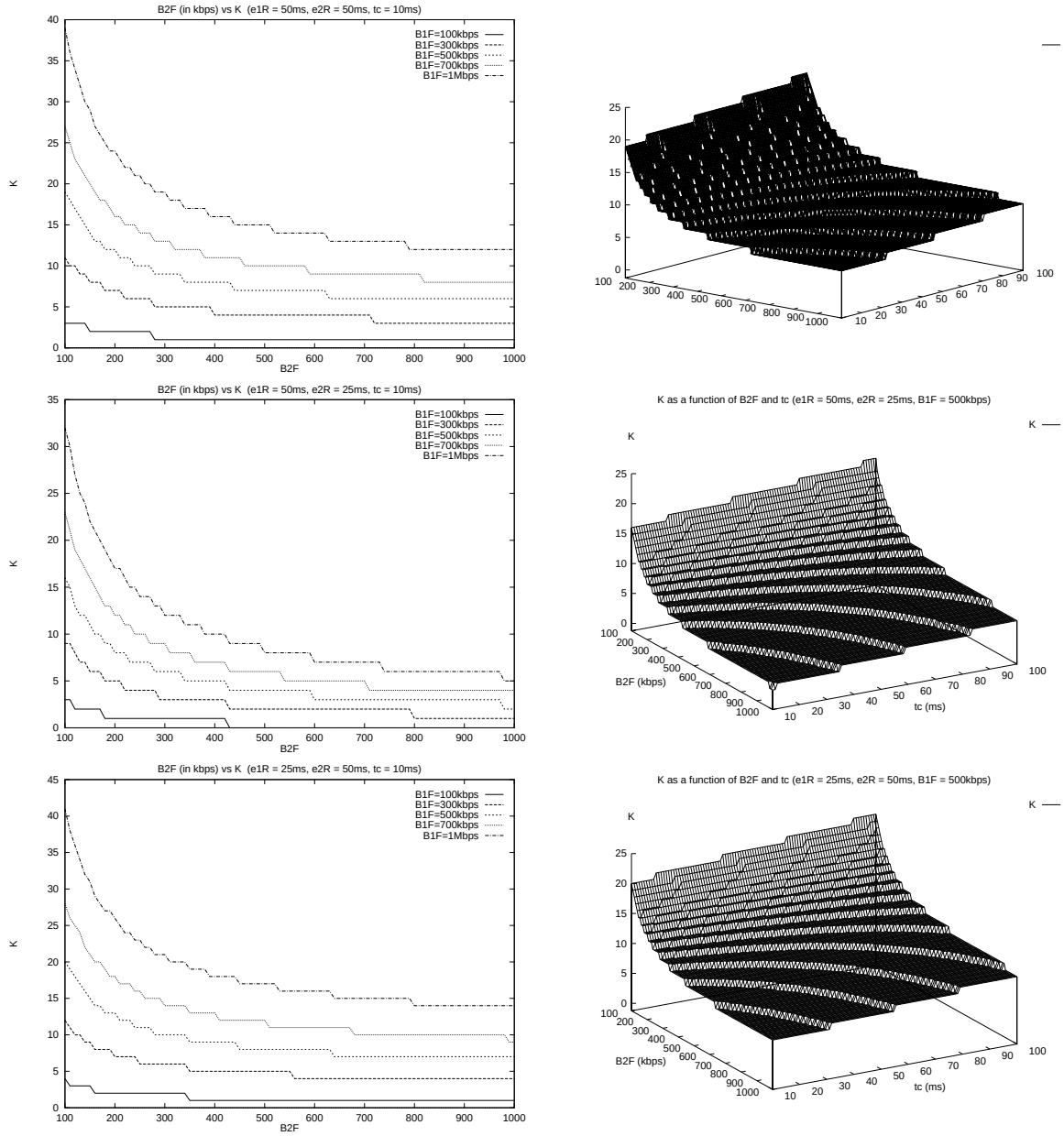


Figure 6.5: Graphing K analytically: 100kbps $\leq B_{1F}, B_{2F} \leq$ 1Mbps

the time of changeover, there will be unnecessary retransmissions. Note that smaller the value of K, the higher the possibility of unnecessary retransmissions, and vice-versa.

In Figure 6.5.2, K varies between 0 and 30, and mostly has a value below 10. Remember that the smaller K is, the more unnecessary retransmissions will occur, and the more cwnd grows when it should not. Changes in e_{1R}, e_{2R} and t_c seem to have little influence on K, as compared to the variation due to B_{1F}, B_{2F}. That is because in this set, since the available bandwidths are low, the total delay is dominated by transmission delay.

In Figure 6.5.2, K varies between 0 and 40. The median value of K in this set has increased from the first set. This increase can be attributed to the greater range of the bottleneck bandwidths. Another important factor can be understood by considering equation (6.8). With an increase in the bottleneck bandwidth, the value of d_{1F} decreases, consequently increasing K. We also observe the increased influence of e_{1R}, e_{2R} and t_c in this set of results, since the transmission delay is lesser dominant in this set.

In both sets, we note that K decreases with a decrease in B_{1F} or an increase in B_{2F}, as is expected.

6.6 Solutions

The cwnd overgrowth and unnecessary retransmissions during changeover can be seen to occur due to inadequacies of SCTP—either (i) the sender is unable to distinguish between SACKs for transmissions and SACKs for retransmissions, or (ii) the congestion control algorithm at the sender is unaware of the occurrence of a changeover, and hence is unable to identify reordering introduced due to changeover. Addressing either of these inadequacies will solve the more important problem of cwnd overgrowth. In [39] and [40], we proposed three solutions to solve the problem of cwnd overgrowth.

- The *Rhein* algorithm [39] addresses (i) above and proposes the addition of two new

chunks called the *Retransmission Identifier (RTID) Chunk* and the *Retransmission Identifier (RTID) Echo*. The RTID chunk is added to every outgoing data packet at the sender, and carries one bit per TSN in the packet. The bit is 0 if its respective TSN is a first transmission, and is 1 if the TSN is a retransmission. The receiver echoes back these bits in the RTID echo chunk in the corresponding ack. Care is taken to ensure the RTID information is stored across delayed acks and such. To avoid cwnd overgrowth due to ambiguity in the SACKs, *Karn's algorithm* is applied to cwnd growth. In other words, a TSN which has been retransmitted does not cause any cwnd growth.

- Two kinds of *Changeover Aware Congestion Control* algorithms that address (ii) above—the *Conservative CACC (C-CACC)*, and the *Split Fast Retransmit CACC (SFR-CACC)*. The key idea in the CACC algorithms is to maintain state at a sender on a per-destination basis when a changeover happens. On receipt of a SACK, the sender uses this state to selectively increase missing report count for TSNs in the retransmission queue. The key difference between the two variants is that C-CACC is simpler but conservative about marking TSNs, while SFR-CACC is more complex but allows marking over a wider range of TSNs.

Of these algorithms, we recommended the SFR-CACC algorithm in [40]. As CMT work progressed, we discovered that the problems discussed in this chapter shared the same cause as unnecessary retransmissions with CMT (Section 2.2). With additional insights gained later, the SFR-CACC algorithm was significantly simplified and the new incarnation was applied to CMT as the SFR algorithm (Figure 2.3).

We currently recommend the newer SFR algorithm as a solution to eliminate unnecessary retransmissions and cwnd overgrowth during changeover. We are currently proposing this algorithm modification to SCTP as an Internet Draft (draft-iyengar-sctp-cacc-03.txt) at the IETF [38].

Chapter 7

DISCUSSION, FUTURE WORK AND RELATED WORK

In this concluding chapter, we first discuss CMT design and deployment considerations in Section 7.1. We then present several ideas for continued research in CMT in Section 7.2 followed by a discussion of related work in Section 7.3. Section 7.4 concludes this dissertation.

7.1 Discussion

This section discusses considerations in CMT design and CMT's potential impact in environments that are not explored in this dissertation.

7.1.1 Alternative Design – Separate Sequence Spaces

Another approach to accomplishing CMT would be to define a *separate sequence space per destination*—a solution often considered the simplest design for CMT [6, 7]. While this solution simplifies some issues, it also introduces its own complications.

- What sequence number is used for a packet that is retransmitted to a destination other than the original? What happens to the sequence number used for the original destination (is it reused, or is it discarded thereby introducing a gap?) Any solution will likely require additional reliable signaling between sender and receiver.

- During association closure, the final sequence number must be agreed upon by sender and receiver to ensure complete reliable transfer. Introducing multiple sequence number spaces complicates this issue.
- Several mechanisms are understood with a single sequence space, for example, reneging. Managing per destination sequence numbering for these mechanisms requires careful examination.
- Separating sequence spaces causes separation of ack info per path. This separation cannot provide CMT's increased resilience to reverse path loss and reverse path failure as shown in Section 3.4.

We believe that the complexities introduced by such a design outweigh the benefits.

7.1.2 Retransmission Timer Calculations

In SCTP, acks for transmitted TSNs are used to estimate the roundtrip time (RTT), which is subsequently used in calculating the retransmission timeout (RTO) [65]. Unexpected behavior can occur with CMT when the paths used have different delay characteristics (see Figure 7.1). Assuming traffic on both paths, an ack sent later on the faster return path (SACK with cumack x+n) may reach the sender sooner than an ack sent earlier on the slower return path (SACK with cumack x). Such later acks which are received sooner, being cumulative, will cause the sender to a different value for RTT than the actual RTT for Path $A_1 - B_1$.

Before addressing the question of whether spurious timeouts will occur with CMT, we first raise the question: What is the RTT when multiple return paths exist, as is the case with CMT? A sender records the time from sending data to a certain destination, until the receipt of the corresponding ack as the RTT for that path. When a single path is used in both directions (as in SCTP or TCP), this recorded time represents (in the general case)

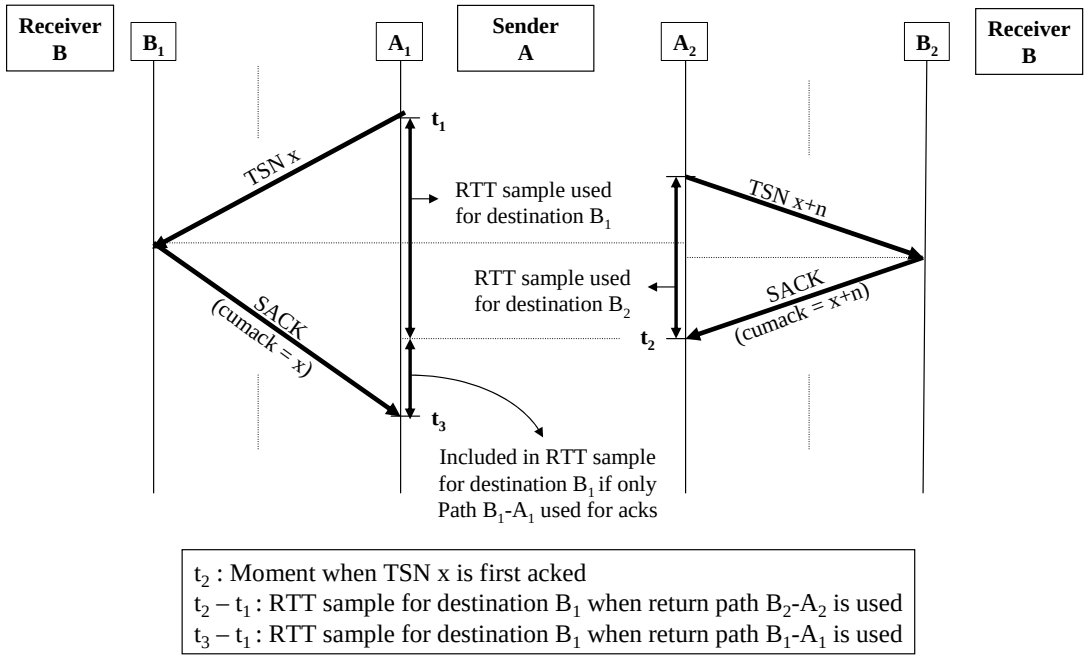


Figure 7.1: RTT Calculation at CMT Sender

the sum of the end-to-end delays on the single forward path and the single return path. When multiple return paths are used for transmitting cumulative acks, which return path is the "roundtrip", and hence, which return path's delay should be used in the sender's RTT estimate?

We suggest that when multiple return paths are used, the estimated RTT at a CMT sender should be a weighted average of the different RTT measures, with the weights reflecting the number of samples. Such an average will reflect the expected ack arrival time for a transmitted data PDU, and is achieved naturally by SCTP's (and TCP's) current RTT estimation mechanism [54, 65], which is used by CMT.

We now discuss the effects of these RTT estimates on the calculated RTO value. Equation (7.1) shows the RTO calculation used in SCTP for loss recovery [65].

$$RTO = SRTT + 4 * RTTVAR \tag{7.1}$$

where SRTT is the average estimated RTT, and RTTVAR is the variance observed in the RTT samples.

The RTO value, thus, accounts for variance observed in the RTT samples. In Figure 7.1, at steady state, with a continuous stream of data and acks on both paths, the sender (host A) will receive acks on both A_1 and A_2. Therefore, when data and ack flows exist on both paths, for data sent on forward path $A_1 - B_1$, some acks will be received on return path $B_1 - A_1$, providing larger RTT samples of $(t_3 - t_1)$, and other acks will be received on return path $B_2 - A_2$, providing smaller RTT samples of $(t_2 - t_1)$. The variance of the RTT samples will, therefore, be larger than when acks are received on only one of the two return paths. This increased variance will consequently cause the RTO value to be larger (and therefore, more conservative), and will prevent spurious timeouts from occurring.

We note that the use of multiple return paths is not unique to CMT—a similar situation arises when acks on the return path in an SCTP (or TCP) association are distributed by a load-balancing router in the network over two paths with different delays.

7.1.3 Applicability With a Shared Bottleneck

This dissertation operates under the strong assumption that the bottleneck queues (in other words, points of congestion) on the end-to-end paths used in CMT are independent. Overlap in the paths is acceptable as long as the paths' bottlenecks are independent. If used in the presence of a shared bottleneck, CMT using n paths will be as aggressive as n separate SCTP associations sharing the bottleneck. An improved CMT sender would be able to detect the presence of a shared bottleneck and respond appropriately, and should be no more aggressive than a single SCTP association through the shared bottleneck. Later in this chapter (Section 7.2.2), we discuss how related work on end-to-end shared bottleneck detection can be applied to CMT. In this section, we discuss the applicability of this dissertation's results when a bottleneck is shared across the paths used for CMT.

- CMT algorithms (Chapter 2) are applicable when reordering due to different path delays occurs within a CMT association. Path delays can be different even if a bottleneck is shared, and thus our algorithms will continue to apply in the presence of a shared bottleneck.

- CMT retransmission policies (Chapters 3 and 4) show differences in performance as the paths' loss rates diverge (for instance, see Figure 4.4). If a bottleneck is shared, the loss rates (due to congestive losses) of the paths sharing the bottleneck should be the same, and therefore the retransmission policies should all perform similarly. If non-congestive losses exist on the paths, the rates of such losses could be different, and a loss-rate-based policy may still perform better than the other policies. If the congestive and non-congestive losses can be tracked separately, then newer loss-rate-based policies that consider both kinds of loss rates should be considered.

- Rbuf blocking (Chapter 4) degrades performance increasingly with increasing loss rate and delay difference between the paths. With a shared bottleneck, while loss rates (due to congestive losses) of the paths sharing the bottleneck should be the same, the delays can still be different, causing rbuf blocking. Further, while the extent may or may not be significant, different rates of non-congestive loss on the paths can also contribute to increased rbuf blocking.

7.1.4 CMT in Other Environments

Small file transfers: Small file transfers (web transfers) suffer from the problem that they are more prone to timeouts because the number of packets in the transfer may be insufficient to trigger a fast retransmission. We have not tested CMT behavior with small files, but note the following. Spreading a small file transfer over multiple paths further decreases the ability for fast retransmit to discover loss, and thus will decrease expected

throughput. At the same time, using the aggregated bandwidth of multiple paths should tend to increase throughput. We suspect that the performance degradation due to timeouts may dominate with small file transfers using CMT.

Failure scenarios: SCTP uses k consecutive timeouts as an indication of failure (recommended value of k is 6 [65]). CMT's failure detection/response mechanisms and latency are currently the same as those of SCTP. In the presence of failure, we observed that CMT's behavior is the same as that of SCTP with a failed primary path (brief transmission periods followed by long silence periods). We believe that further optimization is possible to improve CMT performance during failures, and is part of our future work. In Section 7.2.1, we discuss a promising optimization currently being explored by this author's colleague in the Protocol Engineering Lab.

TCP-to-SCTP translation shim: While the SCTP specification is maturing through the IETF standardization process, a gradual migration path is needed for SCTP deployment. A transparent TCP-to-SCTP translation shim layer proposed by Bickhart [12] provides a mechanism for this migration. Bickhart proposes a shim layer to intercept system calls to TCP and translate them to equivalent SCTP calls, thereby enabling use of an SCTP association in place of a TCP connection. Such a shim allows legacy TCP applications to use SCTP, and benefit from its multihoming features, transparent to the application user. If both endpoints do not support SCTP, the shim falls back to using TCP. CMT can be used and enabled in such a shim, thereby providing improved throughput to even legacy TCP applications running on multihomed endpoints.

7.2 Future Work

We now discuss two directions for future work.

7.2.1 Considerations For Path Failures

When one path used in CMT experiences failure, data outstanding on the failed path has to be recovered through a timeout, resulting in rbuf blocking for the period of the time-out. After a timeout recovery, a sender will continue to send new data to the failed path until *Path.Max.Retransmit (PMR)* consecutive timeouts occur. With the current SCTP recommendation of PMR = 5, a failure results in almost 61 seconds of rbuf blocking. Using PMR = 3 as recommended by Caro [16] can still cause rbuf blocking of upto 15 seconds. We believe that this blocking can be reduced with better management of data transmissions when failure seems imminent, i.e., when a timeout occurs.

This author co-developed an idea with Preethi Natarajan for better handling of failure conditions with CMT.

In addition to the existing *ACTIVE* and *INACTIVE* states for a destination, we introduce a new state—"*POSSIBLY FAILED (PF)*". When a CMT sender experiences a timeout for a destination, the destination is marked *PF*, and the sender should stop sending any data (transmissions or retransmissions) to that destination. Only HeartBeat (HB) probes are sent to a destination in the *PF* state until the destination responds. New data and retransmissions continue to be sent to the other destinations. Thus, the sender experiences rbuf blocking of atmost 1 RTO and then continues to send data on all other paths. Note that HBs sent to a *PF* destination follow the same timeout pattern as data, i.e., HBs follow the more aggressive data timer, and not the slow HB timer. Initial studies have shown that this simple idea seems to work well in handling failure conditions with CMT. This idea is currently being pursued by Preethi to improve CMT performance under failure conditions.

7.2.2 Shared Bottleneck Detection and Response

A *sub-association flow* is defined as a flow within a CMT association consisting of packets having the same source and destination IP addresses. In distributing transmitted data to multiple destination addresses, a CMT sender creates several sub-association flows within the association. These sub-association flows may share (with each other) the available bandwidth at a bottleneck router.

In this dissertation work, we have operated under the strong assumption that the paths used in CMT have independent bottlenecks, i.e., the sub-association flows within a CMT association do not share any bottleneck. On a network such as the Internet, this assumption may be invalid. If a CMT sender incorrectly assumes independent bottlenecks in the presence of a shared bottleneck, the sender sends more aggressively than the other flows sharing the bottleneck. This aggressive behaviour is due to multiple cwnds evolved and used at a CMT sender for sub-association flows sharing a bottleneck, giving the sender an unfair increase in aggregate throughput. It is only fair to the other flows at the bottleneck, which use a single cwnd, that the CMT sender evolve and use a single cwnd instead of multiple cwnds.

An area of future study is to investigate shared bottleneck detection techniques that can be employed by CMT, and response mechanisms for a CMT sender to switch from using multiple cwnds to a single cwnd when independent paths do not exist [5].

Several techniques have been proposed that use non-TCP flows for sender-side detection of a shared bottleneck between end-to-end flows [30, 44, 45, 47, 57]. Further work is needed to determine how these techniques can be employed with a multihomed, window-based, application-limited transport layer sender. The outcome of this research will present simple yet concrete online algorithms for multihomed end-hosts to detect and respond to shared bottlenecks. The larger problem can also be broken down into smaller

components:

1. Evaluation of different end-to-end shared bottleneck detection techniques: Using analysis, simulation and, if possible, emulation, the different techniques need to be evaluated in different topologies and traffic conditions. The goal here is to evaluate the strengths and weaknesses of the different techniques, and to identify what can be used in the context of CMT.

2. Application of these techniques to an application-limited window-based sender: Current techniques do not consider the constraints that a transport layer sender has to operate under, such as application behavior and sending constraints due to congestion control. While some initial work in the area demonstrates feasibility [53], further work is needed to determine how these techniques can be employed with CMT.

3. Investigation of how a sender can seamlessly move between using shared and separate congestion control: A sender must have the ability to seamlessly curb the overall sending rate if a shared bottleneck is detected, and allow the sending rate to quickly ramp up and utilize the available bandwidth when separate bottlenecks are established. While separate cwnds already handle separate congestion control, a larger *aggregate_cwnd*, which tracks the sum of the individual cwnds, can be used when a bottleneck is shared. This variable can control the sending rate of the entire association, while allowing each path to maintain its own cwnd and ack clock.

4. Tradeoffs involved in using multiple paths with shared congestion control: If all bottlenecks are shared, CMT may not gain any throughput benefits over plain SCTP. On the one hand, path delay differences may cause CMT throughput to be lesser than SCTP throughput when all bottlenecks are shared, but on the other hand, CMT may have a faster reaction time if failure occurs. Tradeoffs may thus exist that should be considered when evaluating CMT in the presence of shared bottlenecks.

7.3 Related Work

Load balancing in computer networks is a well studied problem. Though this dissertation is focused on the transport layer, this section provides a survey of load balancing efforts at other layers as well. These other research projects helped identify and understand how common issues have been addressed previously, and provided insight on how to incorporate related ideas. This section broadly classifies related previous work into load balancing at the application, transport, network and link layers, as presented and described in Sections 7.3.1, 7.3.2, 7.3.3, and 7.3.4, respectively.

7.3.1 Load Balancing at the Application Layer

Several applications [29, 59] use multiple TCP connections to increase throughput in high bandwidth networks. These applications load balance over a single path to a receiver in an attempt to maximize use of the large available bandwidth, whereas CMT distributes data over multiple paths. Content Networks [24] provide an infrastructure for *connection level load balancing* at the granularity of TCP connections. CMT provides *T-PDU level load balancing* at the granularity of Transport-PDUs. Connection level load balancing is useful for short TCP connections such as web requests and responses, but can be suboptimal for long bulk data transfers, where the server is constrained to a single path throughout the transfer.

While we are not aware of any instance, an application could conceivably attempt sub-connection scale load balancing. We argue that even if such an application were attempted, load balancing at these T-PDU scales is best done at the transport layer, which, being the lowest end-to-end layer, has the most accurate information about end-to-end path(s). CMT uses loss and delay information for redirection of retransmissions - such decisions are best made in the transport layer. A load balancing application that desires high level of control at T-PDU scales would functionally reside in the transport layer,

and would require significantly increased communication between the transport and the application layers.

We further argue that in general, load balancing at the application layer increases code redundancy and room for error by requiring independent implementations in each application.

7.3.2 Load Balancing at the Transport Layer

mTCP [72], an effort parallel with ours, implements a transport layer solution to aggregate bandwidth across multiple end-to-end paths. mTCP, like CMT, uses a single sequence space across paths. mTCP significantly modifies TCP to use multiple paths provided by an overlay network (RON [9]), and like CMT, employs mechanisms to handle reordering side-effects. mTCP introduces a new shared bottleneck detection mechanism to detect and respond to shared bottlenecks, and proposes a heuristic-based "path suppression" mechanism which suppresses use of paths that have "sufficiently low" throughput. While path suppression avoids throughput degradation in mTCP due to rbuf blocking and path failure, unlike CMT, mTCP does not consider retransmission policies in reducing the effects of rbuf blocking. RON is assumed as the underlying routing layer for mTCP, and is required for obtaining multiple paths; that is, mTCP cannot be used on an arbitrary IP network. On the other hand, CMT leverages native transport layer multihoming mechanisms in SCTP, and can be used on any IP network. mTCP also uses a single return path for ack traffic, thereby requiring additional mechanisms to detect failure of the single ack path, and causing performance degradation during failure. Inspite of the differences between mTCP and CMT, there are significant similarities in their design and the issues considered. Ideas proposed in mTCP, such as shared bottleneck detection/response and path suppression, are broadly applicable to CMT, and are worth evaluating for future incorporation into CMT.

Al et al. [6,7] suggest ideas for *load sharing* that requires additional metadata in the SCTP PDUs. We believe that current SCTP (and TCP-SACK) PDUs already contain sufficient information for the data sender to infer the per-path ordering information that [7] explicitly codes as metadata. Reference [7] fails to suggest modified procedures for mechanisms which are immediately affected, such as initialization of the per-path sequence numbers, association initialization and shutdown procedures with multiple sequence numbering schemes, and response to reneging by a receiver. We have also seen that sharing sequence number space across paths improves performance (see Section 3.4) whereas [7] uses a separate sequence number space per path, and will therefore not see CMT's performance benefits. Further, [7] assumes that the rbuf does not constrain a sender which is unrealistic in practice.

Argyriou et al. [10] provide techniques for *bandwidth aggregation* with SCTP, but do not present and analyze their protocol modifications to SCTP. The modified fast retransmission algorithm is simplistic and assumes information that is not available to an SCTP receiver. For instance, the implicit assumption that a receiver will be able to differentiate a packet loss from reordering is unrealistic.

Hsieh et al. [31] propose *pTCP (parallel TCP)* which provides an infrastructure for data striping within the transport layer. pTCP has two components—Striped connection Manager (SM) and TCP-virtual (TCP-v). The TCP-v's are separate connections that are managed by the SM. TCP-v probes the path and performs congestion control and loss detection/recovery, while the SM decides which data is sent on which TCP-v. This decoupling of functionality avoids some pitfalls of application layer approaches, and allows for intelligent scheduling of transmissions and retransmissions. A significant issue with pTCP is its complexity. As the authors note, maintenance of multiple Transmission Control Blocks at a sender can be a resource sink [31]. Implementation is also complex, since pTCP replicates transport layer functionality such as connection establishment/teardown

and checksum calculations. Further, pTCP has several unresolved issues. If both sender and receiver are multihomed with two IP addresses each, pTCP does not explain how a sender decides on which sender-receiver pairs to establish TCP connections - a complex problem. Plugging transport protocols into pTCP also requires non-trivial modifications to the transport protocols themselves. CMT, on the other hand, uses a transport protocol with built-in mechanisms for multihoming.

7.3.3 Load Balancing at the Network Layer

Phatak and Goff [56] propose distributing data at the network layer transparent to the higher layers using IP-in-IP encapsulation. The authors identify conditions under which this mechanism avoids incorrect retransmission timeouts. The proposed solutions *assume* end-to-end delays are dominated by fixed transmission delay, and do not apply to propagation delay dominated paths, or paths with dynamically changing bandwidths and delays. CMT's algorithms do not require such assumptions, and will operate under dynamic and propagation delay dominated conditions.

Several proposals exist for *multipath routing* - routing packets from a source to a destination network over multiple paths. However, different paths are likely to exhibit different RTTs, thus introducing packet reordering. TCP's performance degrades in the presence of increased reordering. To enable optimal load balancing at intermediate routers without affecting end-to-end TCP performance, modifications to TCP have also been proposed [13, 14, 27, 71]. These proposals augment and/or modify TCP's congestion control mechanisms to cope with reordering introduced by network layer load balancing; the burden of actually using multiple paths in the network is left to the intermediate routers.

On one hand, though there is motivation for the routing infrastructure in the Internet to use multiple paths simultaneously, it is important to ensure *a priori* that the endpoints are capable of handling the reordering introduced. On the other hand, the endpoints will be

able to benefit from these proposals only when the routing infrastucture uses the multiple paths. There is no significant motivation for endpoints to implement and use any of these proposals since performance gain, if any, is not immediate, but long term. These issues, though not strictly research issues, govern the deployment and fruition of the proposals described in this section.

In the Internet, the end user has knowledge of, and control over, only the multihomed end hosts, not the intermediate routers. In such cases the end host cannot dictate or govern use of multiple paths in the network. But the end host can use multiple end-to-end paths available to the host [67], thus motivating CMT at the transport layer.

7.3.4 Load Balancing at the Link Layer

Load balancing research at the link layer, also known as *link layer striping* or *inverse multiplexing*, generally addresses packet resequencing and fair load distribution across a single link. Link layer research addresses the needs of service providers to increase bandwidth by adding more links, and striping data across the links to provide increased aggregated bandwidth, instead of having to replace exisiting infrastucture. Generally, the link characteristics do not vary significantly, and end-to-end congestion control is not a concern, thus reducing the load balancing problem to those of fairness and packet resequencing. Research in inverse multiplexing [4, 23, 68] or link aggregation [60] has generally not been end-to-end, and the operating conditions do not represent the conditions that end-to-end CMT over the Internet has to operate in.

Packet resequencing techniques address packet reordering due to load balancing across multiple links. In synchronous packet resequencing techniques, the receiver and the sender are synchronized to allow the receiver to receive segments in order from its multiple links. In comparison to an environment such as the Internet, such techniques operate within better known delay difference and bandwidth difference bounds. Asynchronous

packet resequencing techniques, such as [21,46] require more metadata, such as sequence numbers or rank information [46] to be added to the packets and/or assume information such as bitrate on the channels. Information such as bitrate and delay are not static and cannot be assumed for an end-to-end path through the Internet. Further, adding metadata is unnecessary for a SACK-based transport protocol since the sender can infer the necessary information from the SACKs.

Snoeren [61] proposes a more general adaptive inverse multiplexing mechanism called Link Quality Balancing (LQM), which uses relative link performance metrics to schedule traffic across logically bundled point-to-point links in Wireless WANs (WWANs). This scheduling mechanism is implemented in Wide-Area Multilink PPP (WAMP), which leaves bandwidth probing and reliability to the higher layers (in [61], to TCP), and focuses only on scheduling of traffic taking link quality metrics into consideration. This work may be useful in future investigation of scheduling mechanisms for CMT.

7.4 Summary

This dissertation investigated and evaluated design considerations in implementing CMT at the transport layer over independent end-to-end paths using SCTP as an example of a multihome-capable transport layer protocol. We proposed algorithms to handle reordering due to CMT, and introduced retransmission policies for CMT. We presented the problem of rbuf blocking, argued that it cannot be eliminated, and demonstrated that intelligent retransmission decisions at the transport layer can reduce performance degradation due to rbuf blocking. We discussed our implementation of CMT in BSD-SCTP, and pointed out potential areas for continued research in CMT.

Varied use and experimentation with CMT in different environments and under different constraints will contribute to a better understanding of CMT and to uncovering of hitherto unknown issues. We, therefore, encourage wider use of, and experimentation with, CMT.

BIBLIOGRAPHY

[1] Future combat systems website.
http://www.globalsecurity.org/military/systems/ground/fcs.htm.

[2] KAME Project. http://www.kame.net.

[3] CAIDA: Packet Sizes and Sequencing, Mar 1998. http://traffi c.caida.org.

[4] H. Adiseshu, G. Parulkar, and G. Varghese. A Reliable and Scalable Striping Protocol. In *ACM SIGCOMM 1996*, Stanford, California, August 1996.

[5] A. Akella, S. Seshan, and H. Balakrishnan. The Impact of False Sharing on Shared Congestion Management. In *ICNP 2003*, Atlanta, GA, November 2003.

[6] A. Abd El Al, T. Saadawi, and M. Lee. Improving Throughput and Reliability in Mobile Wireless Networks via Transport Layer Bandwidth Aggregation. *Computer Networks, Special issue on Military Communications Systems and Technologies*, 46(5), December 2004.

[7] A. Abd El Al, T. Saadawi, and M. Lee. LS-SCTP: A Bandwidth Aggregation Technique For Stream Control Transmission Protocol. *Computer Communications, Special issue on Protocol Engineering for Wired and Wireless Networks*, 27(10), June 2004.

[8] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC2581, IETF, April 1999.

[9] D. Andersen, H. Balakrishnan, and R. Morris M. Kaashoek. Resilient Overlay Networks. In *18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Canada, October 2001.

[10] A. Argyriou and V. Madisetti. Bandwidth Aggregation With SCTP. In *IEEE Globecom 2003*, San Fransisco, CA, December 2003.

[11] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet Reordering is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, 7(6), December 1999.

[12] R. Bickhart. *TCP-to-SCTP Translation Shim*. MS Thesis, CIS Dept, University of Delaware, May 2005.

[13] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM Computer Communication Review*, 32(1), January 2002.

[14] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka. TCP-PR: TCP for Persistent Packet Reordering. In *IEEE ICDCS 2003*, Rhode Island, May 2003.

[15] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC1122, IETF, October 1989.

[16] A. Caro. End-to-End Fault Tolerance Using Transport Layer Multihoming. Phd dissertation, CIS Dept, University of Delaware, August 2005.

[17] A. Caro, P. Amer, J. Iyengar, and R. Stewart. Retransmission Policies with Transport Layer Multihoming. In *ICON 2003*, Sydney, Australia, September 2003.

[18] A. Caro, P. Amer, and R. Stewart. Rethinking End-to-End Failover With Transport Layer Multihoming. *Annals of Telecommunications*. (in press).

[19] A. Caro, P. Amer, and R. Stewart. Retransmission Policies for Multihomed Transport Protocols. *Computer Communications*. (in press).

[20] A. Caro, J. Iyengar, P. Amer, S. Ladha, G. Heinz, and K. Shah. SCTP: A Proposed Standard for Robust Internet Data Transport. *IEEE Computer*, 36(11):56–63, November 2003.

[21] F. M. Chiussi, D. A. Khotimsky, and S. Krishnan. Generalized Inverse Multiplexing of Switched ATM Connections. In *IEEE GLOBECOM 1998*, Sydney, Australia, November 1998.

[22] K. Claffy, G. Miller, and K. Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. *INET 1998*, April 1998.

[23] The ATM Forum Technical Committee. Inverse Multiplexing for ATM (IMA) Specification, Version 1.0, July 1997. AF-PHY-0086.000.

[24] M. Day, B. Cain, G. Tomlinson, and P. Rzewski. A Model For Content Internetworking (CDI). RFC3466, IETF, February 2003.

[25] H. Ekstrom and R. Ludwig. The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport. In *IEEE INFOCOM 2004*, Hong Kong, March 2004.

[26] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC2883, IETF, July 2000.

[27] M. Gerla, S. S. Lee, and G. Pau. TCP Westwood Simulation Studies in Multiple-Path Cases. In *SPECTS 2002*, San Diego, California, July 2002.

[28] K. D. Gradisching and M. Tuexen. Signalling Transport Over IP-based Networks using IETF Standards. In *Third International Workshop on Design of Reliable Communication Networks (DRCN 2001)*, Budapest, Hungary, October 2001.

[29] T. Hacker and B. Athey. The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network. In *IEEE IPDPS*, Ft. Lauderdale, FL, April 2002.

[30] K. Harfoush, A. Bestavros, and J. Byers. Robust Identification of Shared Losses Using End-to-End Unicast Probes. In *ICNP 2000*, Osaka, Japan, October 2000.

[31] H.Y. Hsieh and R. Sivakumar. A Transport Layer Approach for Achieving Aggregate Bandwidths on Multihomed Mobile Hosts. In *ACM International Conference on Mobile Computing and Networking (MOBICOM)*, Atlanta, Georgia, September 2002.

[32] S. Iren, P. Amer, and P. Conrad. The Transport Layer: Tutorial and Survey. *ACM Computing Surveys*, 31(4), December 1999.

[33] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking*. (in press).

[34] J. Iyengar, P. Amer, and R. Stewart. Performance Implications of a Bounded Receive Buffer In Concurrent Multipath Transfer. Tech. Report, CIS Dept., University of Delaware.

[35] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using Transport Layer Multihoming: Performance Under Varying Bandwidth Proportions. In *MILCOM 2004*, Monterey, CA, October 2004.

[36] J. Iyengar, P. Amer, and R. Stewart. Retransmission Policies For Concurrent Multipath Transfer Using SCTP Multihoming. In *ICON 2004*, Singapore, November 2004.

[37] J. Iyengar, P. Amer, and R. Stewart. Receive Buffer Blocking In Concurrent Multipath Transport. In *IEEE GLOBECOM*, St. Louis, Missouri, November 2005.

[38] J. Iyengar, P. Amer, R. Stewart, and I. Arias-Rodriguez. Preventing SCTP Congestion Window Overgrowth During Changeover. draft-iyengar-sctp-cacc-03.txt, Internet Draft, IETF, November 2005.

[39] J. Iyengar, A. Caro, P. Amer, G. Heinz, and R. Stewart. SCTP Congestion Window Overgrowth During Changeover. In *SCI 2002*, Orlando, FL, July 2002.

[40] J. Iyengar, A. Caro, P. Amer, G. Heinz, and R. Stewart. Making SCTP More Robust to Changeover. In *SPECTS 2003*, Montreal, Canada, July 2003.

[41] J. Iyengar, K. Shah, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming. In *SPECTS 2004*, San Jose, California, July 2004.

[42] N. Jani and Krishna Kant. SCTP Performance in Data Center Environments. Technical report, Intel Corporation, 2005.

[43] A. Jungmaier. SCTP For Beginners, 2001.
http://tdrwww.exp-math.uni-essen.de/inhalt/forschung/sctp_fb/.

[44] D. Katabi, I. Bazzi, and X. Yang. A Passive Approach for Detecting Shared Bottlenecks. In *IEEE ICCCN 2001*, October 2001.

[45] D. Katabi and C. Blake. Inferring Congestion Sharing and Path Characteristics from Packet Interarrival Times. Technical report, MIT Lab for Computer Science, 2001.

[46] D. A. Khotimsky. A Packet Resequencing Protocol for Fault-Tolerant Multipath Transmission with Non-Uniform Traffic Splitting. In *IEEE GLOBECOM 1999*, Rio de Janeiro, Brazil, December 1999.

[47] M. S. Kim, T. Kim, Y. Shin, S. S. Lam, and E. J. Powers. A Wavelet-based Approach to Detect Shared Congestion. In *ACM SIGCOMM*, Portland, Oregon, August 2004.

[48] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. Technical report, ICIR, 2004.

[49] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). draft-ietf-dccp-spec-13.txt, December 2005. (work in progress).

[50] S. Ladha, S. Baucke, R. Ludwig, and P. Amer. On Making SCTP Robust to Spurious Retransmissions. *ACM SIGCOMM Computer Communication Review*, 34(2):123–135, April 2004.

[51] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the Self-similar Nature of Ethernet Traffic. In *ACM SIGCOMM 1993*, San Francisco, CA, September 1993.

[52] R. Ludwig and R. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communications Review*, 30(21):30–36, January 2000.

[53] Pavlos Papageorgiou and Michael Hicks. Merging Network Measurement with Data Transport (Extended Abstract). In *IEEE Passive/Active Measurement Workshop (PAM)*, Boston, MA, March 2005.

[54] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC2988, IETF, November 2000.

[55] B. Penoff and A. Wagner. Towards MPI Progression Layer Elimination With TCP and SCTP. In *Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS): Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes, Greece, April 2006.

[56] D. S. Phatak and T. Goff. A Novel Mechanism for Data Streaming Across Multiple IP Links for Improving Throughput and Reliability in Mobile Environments. In *IEEE INFOCOM 2002*, New York, NY, June 2002.

[57] D. Rubenstein, J. Kurose, and D. Towsley. Detecting Shared Congestion of Flows Via End-to-End Measurement. *IEEE/ACM Transactions on Networking*, 10(3), June 2002.

[58] S. Shakkottai, R. Srikant, A. Broido, and k. claffy. The RTT Distribution of TCP Flows in the Internet and its Impact on TCP-based Flow Control. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), February 2004.

[59] H. Sivakumar, S. Bailey, and R. Grossman. PSockets: The Case For Application-Level Network Striping For Data Inttensive Applications Using High Speed Wide Area Networks. In *IEEE Supercomputing (SC)*, Dallas, TX, November 2000.

[60] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti. The PPP Multilink Protocol (MP). RFC1990, IETF, August 1996.

[61] A. C. Snoeren. Adaptive Inverse Multiplexing for Wide-Area Wireless Networks. In *IEEE GLOBECOM 1999*, Rio de Janeiro, Brazil, December 1999.

[62] W. Stevens, B. Fenner, and A. Rudoff. *Unix Network Programming: Volume 1*. Addison-Wesley, 2004.

[63] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues. draft-ietf-tsvwg-sctpimpguide-16.txt, October 2005. (work in progress).

[64] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison Wesley, New York, NY, 2001.

[65] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC2960, October 2000.

[66] T. Strayer and A. Weaver. Evaluation of Transport Protocols for Real-time Communications. Technical Report TR-88-18, CS Dep., University of Virginia, June 1988.

[67] R. Teixeira, K. Marzullo, S. Savage, and G.M. Voelker. In Search of Path Diversity in ISP Networks. In *USENIX/ACM Internet Measurement Conference*, Miami, FL, October 2003.

[68] C.B.S. Traw and J. M. Smith. Striping Within the Network Subsystem. *IEEE Network*, 9(4):22–32, July 1995.

[69] J. Walrand. *Communication Networks: A First Course*. Aksen Associates, 1991.

[70] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-Similarity Through High-Variability: Statical Analysis of Ethernet LAN Traffic at the Source Level. In *ACM SIGCOMM 1995*, Cambridge, MA, August 1995.

[71] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *ICNP*, November 2003.

[72] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. In *USENIX*, June 2004.